f·i·d·ĕ·™

Fuzzy
Inference
Development
Environment

REFERENCE MANUAL

Contents

Part III Aptronix Run Time Library

Chapter 1 Preliminary

Overview

FIL (Fide Inference Language) is a language for describing units in a fuzzy inference system. The description of a unit in FIL is the source statements of that unit. The units currently supported are the Fide Inference Unit, the Fide Operation Unit and the Fide Execution Unit. They are abbreviated as FIU, FOU and FEU, respectively.

As with any language, FIL has its specific syntax and semantics. Defining FIL means giving a complete description of its syntax and semantics. The semantics of FIU is based on the mathematic definition of a fuzzy inference unit, which is described in Chapter 2. Subsequent chapters present descriptions of the syntactic formulas and their semantics. Although Backus-Naur formulas are presented, the English descriptions are self-sufficient. Readers who are not familiar with the syntactic formulas can refer to the English description alone.

The syntactic formulas of FIL are collected in Appendix A for quick reference. The rule macros of FIU are presented in Appendix D.

Morphologic Terms

The following terms are used in the ordinary sense and are not defined formally:

Identifier

An identifier is a sequence of letters and/or digits in which the first character is a letter. A letter is either a lower case letter (from a to z) , an uppercase letter (from A to Z) or the underscore _ character. An uppercase letter is considered different from the corresponding lowercase letter. (FIL is case sensitive.)

The semantic role of an identifier is as a name of a variable or a label, depending on the syntactic position.

Number

A number is a sequence of decimal digits with or without a decimal point, and possibly prefixed by a plus sign or a minus sign.

The semantic role of a number is as a grade, a value related to a variable, or a grade precision, depending on the syntactic position.

String

A string is a sequence of characters included in a pair of double quotation marks. (Any reserved word or symbol within a string loses its special meaning.)

The semantic meaning of a string is either an engineering unit or a filename.

Reserved words and symbols

A reserved word is a sequence string of letters that plays a predefined syntactic role and is therefore not allowed to be used as an identifier. Each reserved word has a syntactic name, which is to be used in the definition of FIL syntax. Some reserved words have a symbolic form as well. In its symbolic form, a reserved word is represented by a special character. Whenever a reserved word having a symbolic form occurs, it can be replaced by its corresponding symbol (i.e. its symbolic form).

The following is a complete list of the reserved words, their names and their symbolic forms.

Table 1.1 Reserved words

Syntactic name	word	symbol
invarSymbol	invar	>
outvarSymbol	outvar	<
atSymbol	at	@

ofSymbol	of	%
sameSymbol	same	=
copySymbol	copy	!
cutSymbol	cut	
shiftSymbol	shift	~
concSymbol	conc	#
minSymbol	min	&
maxSymbol	max	|
sumSymbol	sum	+
prodSymbol	prod	*
centroidSymbol	centroid	*
maximizerSymbol	maximizer	^
ifSymbol	if	!
thenSymbol	then	:
andSymbol	and	&
isSymbol	is	=
endSymbol	end	.
fiuSymbol	fiu	
fouSymbol	fou	
feuSymbol	feu	
callSymbol	call	
calcSymbol	calculate	
tableSymbol	table	
tvfiSymbol	tvfi	
mamdaniSymbol	mamdani	
bunionSymbol	bunion	
binterSymbol	binter	
leftParenthesisSymbol		(
rightParenthesisSymbol		)
leftBracketSymbol		[
rightBracketSymbol		]
leftBraceSymbol		{
rightBraceSymbol		}
commaSymbol		,
semicolonSymbol		;
colonSymbol		:
quotationMarkSymbol		"

dollarSignSymbol	$
plusSymbol	+
minusSymbol	-
timesSymbol	*
divideSymbol	/
underscoreSymbol	_
doubleunderscoreSymbol	__
leftSymbol	-
rightSymbol	+

Note: Some symbols are shared by more than one reserved words. However, no confusion will be caused thereby, because the reserved words must be used in different contexts.

The above symbols will be regarded as terminate symbols in the syntactic formulas and their syntactic names are used in the formulas.

Comment

A dollar sign starts a one line comment. More precisely, whenever a dollar sign occurs, the segment between the dollar sign and the end of the same line will be treated as being blank.

Minimal Semantics Versus Official Semantics.

Two different but related semantics are presented: the minimal semantics and the official semantics. The minimal semantics is used in Fide converter and emulator, for different target engines. The details of the minimal semantics are hardware related. The official semantics is used for Fide debugger and library functions. The adjective "official" is used before grammar terms which are meaningful only in official semantics.

The major differences between the minimal and official semantics are:

(1) The grade level set, the ranges of variables and the membership functions in the official semantics are approximation of the ones specified in the source code.

(2) The official semantics allow more variations in specifying membership functions.

Chapter 2 Fuzzy Inference Unit

Fuzzy Inference Unit Definitions

Definitions relating to fuzzy inference units are given in this chapter.

A fuzzy inference unit consists of a fuzzy inference chart and a list of official membership functions. The list of official membership functions is used in the official semantics.

Fuzzy inference chart

A fuzzy inference chart (Figure. 2.1) consists of nodes and arrows, and has several attributes: the inference method, the logical operators and the grade representation.

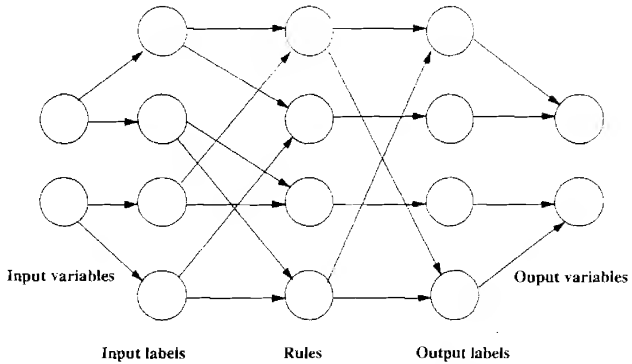


Figure 2.1 *A fuzzy inference chart consists of nodes and arrows*

The nodes are divided into five groups: (1) input variable nodes, (2) input label nodes, (3) rule nodes, (4) output label nodes, and (5) output variable nodes.

They are also called input variables, input labels, rules, output labels and output variables, respectively, for short.

An arrow links a source node and a destination node. It is said to be an outgoing arrow of the source node and the incoming arrow of the destination node (Figure. 2.2). The source node and the destination node are said to be linked to each other.

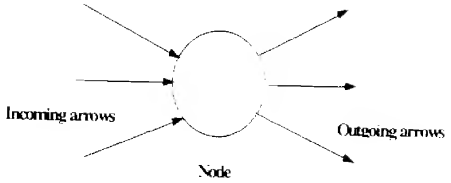


Figure 2.2 *Incoming and outgoing arrows*

It is required that: (1) An input variable has no incoming arrow, but has several outgoing arrows linking the input variable to input labels; (2) An input label has exactly one incoming arrow linking the input label to an input variable, and several outgoing arrows linking the input label to rules; (3) A rule has several incoming arrows linking the rule to input labels, and several outgoing arrows linking the rule to output labels; (4) An output label has several incoming arrows linking the output label to rules, and exactly one outgoing arrow linking the output label to an output variable; and (5) An output variable has several incoming nodes linking the output variable to output labels, but has no outgoing arrow.

A node other than a rule is either of G-type (grade type) or V-type (value type). It is required that: (1) a G-type input variable has exactly one outgoing arrow linking the input variable to a G-type input label; and (2) A G-type output variable has exact one incoming arrow linking the output variable to a G-type output label (Figure 2.3).

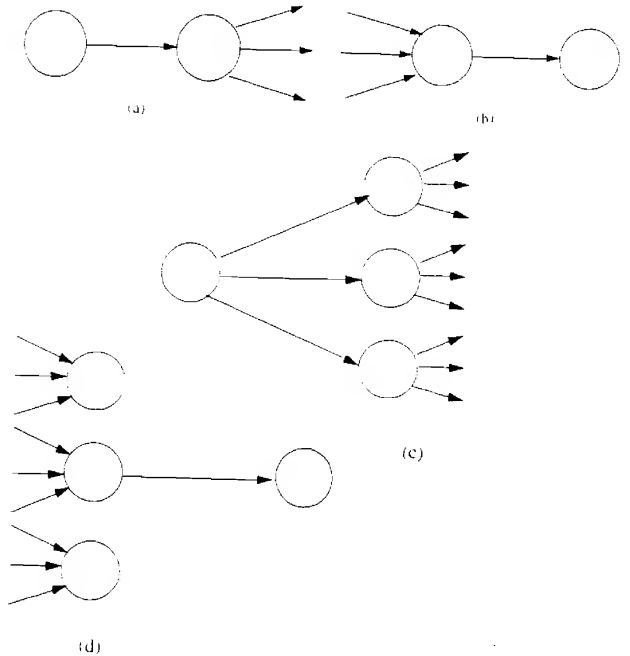


Figure. 2.3 *Variables of different types: (a) G-type input variable, (b) G-type output variable, (c) V-type input variable, (d) V-type output variable.*

Inference method

A fuzzy inference chart has an inference method as one of its attributes. The inference method of the fuzzy inference chart is either the truth value flow inference (TVFI) method or Mamdani's method. These methods are represented by inference symbols as listed below:

Table 2.1 Inference symbols

Symbol	Meaning
tvfi	Truth Value Flow Inference (TVFI)
mamdani	Mamdani method

The default inference method is the TVFI method.

Logical operators

A fuzzy inference chart has a logical conjunction operator and a logical disjunction operator as two of its attributes.

The two operators are selected from the following: minimization operator, maximization operator, the probability product operator, the probability sum operator, the boundary union operator and the boundary intersection operator. The operators are represented by symbols listed below:

Table 2.2 Logical operations

Operation	Meaning
min(x,y)	minimization
max(x,y)	maximization
prod(x,y)	probability product: x*y
sum(x,y)	probability sum: x + y - prod(x,y)
bunion(x,y)	boundary union: min(x+y,1)
binter(x,y)	boundary intersection: max(1-x-y,0)

The default logical conjunction and disjunction operators are minimization and maximization operators, respectively.

Grade representation

A fuzzy inference unit has a grade representation as one of its attributes. The grade representation consists of two components: the internal precision and the external format.

The internal precision specifies how the grades (or truth values) are represented during the inference process. FIL assumes binary fixed-point representation of grade values in the inference process. Therefore, the internal precision is defined as the number of bits used in the representation. The internal grade precision can be any integer between 4 and 16.

Let the internal precision be p. Then the grades are represented by an integer in the integer interval [0,M], where M is the largest integer with no more than p bits (or, equivalently, the largest integer less than the p-th power of two). This interval is called the grade level set. An integer in this interval is called a grade level and M is called the maximal grade level. The grade level corresponding to a given grade (a real number in the interval [0,1]) is called the level of that grade. More precisely, the level of a grade g is equal to the integer part of the product of M and g.

For example, if p=8, then M=255, and the level of grade value 0.3 is the integer part of 255 x 0.3 = 76.

The external format specifies how the grades are represented in the source code. There are two options for the external format: (1) a mathematic form, i.e. a grade in the source code is written as a number in the real interval [0,1], and (2) a level form, i.e. a grade in the source code is written as an integer number in the integer interval [0,M].

In this document, a grade represented in the mathematic form is called a grade value, and the grade level corresponding to the grade value is called the level of the grade (value). In the above example, the level of the grade value 0.3 is 76.

While applied to grade levels, the logical operations are defined as listed in table 2.3.

Table 2.3 Logical operations applied to grade levels

Operation	meaning
min(x,y)	minimization
max(x,y)	maximization
prod(x,y)	x * y / 255
sum(x,y)	x + y - prod(x,y)
bunion(x,y)	min(x+y,255)
binter(x,y)	max(255-x-y,0)

Variable

A variable has several attributes: a name, an engineering unit, a range, an official range, and, if the variable is a V-type output variable, a defuzzifier.

variable name

The name of a variable is an (external) identifier of the variable, i.e., no two variable have the same name.

The names play no role during the inference process.

engineering unit

The engineering units such as "mph" or "grams per cubic centimeter" can be regarded as a string of characters and attached to a V-type variable as an (external) attribute.

The engineering units play no rule during the inference process.

The default engineering unit is an empty string of characters.

range

The range of a variable specifies how its values are represented.

A range consists of a minimum, a maximum and a step length (Figure. 2.4).

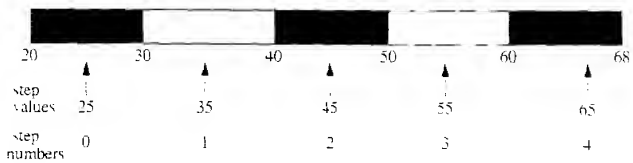


Figure. 2.4 ***A range with minimum = 20, maximum = 68 and step length = 10. The steps are shown in intervals painted alternatively black and white.***

The values of the corresponding variable are supposed to be in the interval between the minimum and the maximum. The interval is called the range interval and is further divided into subintervals called steps. All but the right-most steps of a range should have the same length, the step length (the right-most step can be shorter). A value in the range interval which is equal to the sum of a half step length and the value of the left end-point of a step, is called a step value. The set of all step values is called the step value set of the range. The (algebraic) difference from one step value to another is called a distance value.

The step values are numbered naturally from left to right, starting from 0. The ordinal numbers of a step is called the step number of the step values. There is a one-to-one correspondence between the step values and the step numbers.

The minimum, maximum, step length, range interval, steps, step values, step values set, step numbers, step number set and distance values of a range of a variable are said to be "of the variable". These terms are to be understood in this context for the following sections.

Internally, a value of the variable is represented by the step number of the step value closest to the value (the right one, if two step values are found at the same distance).

The default range of a V-type variable consists of:

minimum = 0
maximum = 255
step length = 1/256

The default range of a G-type variable depends on the grade representation of the FIU. Let M be the maximal grade level. If the mathematic form is used in the representation:

minimum = 0
maximum = 1
step length = 1/(M+1)

Otherwise, if the level form is used:

minimum = 0
maximum = M
step length = 1

No range other than the default one can be attached to a G-type variable.

defuzzifier

A defuzzifier specifies the method of defuzzification.

The possible defuzzifiers are: (1) the centroid defuzzifier; (2) the maximization defuzzifier; (3) the left-most maximization defuzzifier; and (4) the right-most maximization defuzzifier.

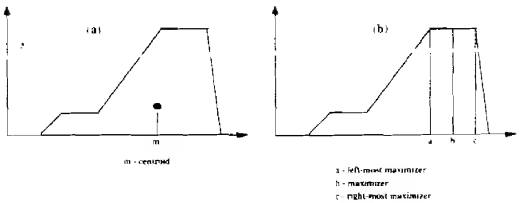


Figure. 2.5 ***Defuzzifiers (a) centroid defuzzifier, (b) maximizer defuzzifiers.***

label

A V-type label has a name and a membership function as its attributes.

label name

The name of a label is an (external) identifier of the label among those linked to the same variable, i.e., no two labels linked to the same variable have the same name. The names play no role during the inference process.

membership function (MBF)

Let S be the step value set S of a variable. The MBF of a label linked to the variable is a function mapping S to the grade level set. The function maps S to the integer interval [0,M], where M is the maximal grade (Figure. 2.6).

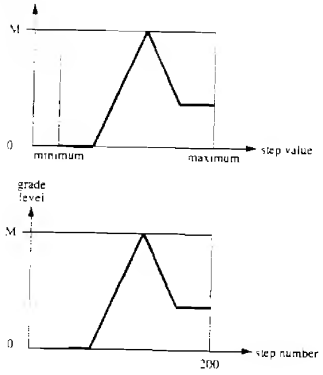


Figure. 2.6 ***(a) A membership function is a mapping from the step value set to the grade level set (b) The same membership function regarded as a mapping from the step number set to the grade level set (the largest step number is supposed to be 200).***

A MBF is said to be a singleton, if there is a step value v in the set S such that, for all u in S, f(u) is non-zero if and only if u=v, and f(v)=M. The step value v is said to be the support point of the MBF (Figure. 2.7).

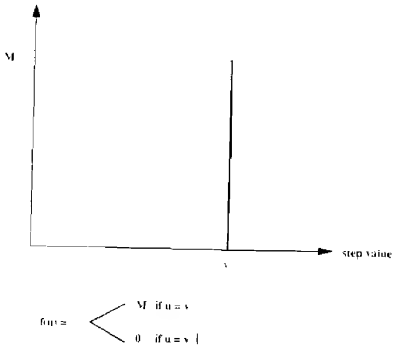


Figure. 2.7 ***A singleton is a degenerated membership function***

If the inference method of the fuzzy inference unit is the TVFI method, then the membership functions of all V-type output labels must be singletons.

Internally, a membership function is regarded as a function mapping the set of the step numbers to the grade level set. The correspondence between the membership function and its internal representation is trivial (Figure. 2.6).

Official membership function list

The official membership function list is a list official membership functions.

Official grade representation

Let p be the internal precision of the fuzzy inference unit. And q be the minimum of p and 8. Substituting q for p in the grade representation, we obtain another grade representation, called the official grade representation. We will use the term "official grade level" with respect to the official grade representation.

Official membership function

Let G be the set of official grade levels, and I be the integer interval [0,256). An official membership function is a mapping from I to G.

Official membership functions of a variable

Official range of a variable

The official range of a variable is obtained by modifying the range of the variable as described below.

(1) If the number of steps of the range is greater than 256, let the step length be one 255-th of the difference between the maximum and minimum of the range.

(2) If the number of steps of the range is less than 256, let the maximum be the sum of the minimum and 255 times the step length.

The Official membership function of a label

Let V be a variable of V-type, L be a label linked to V, and f be the membership function of L. The official membership function corresponding to f is a

membership function g mapping the official step value set of V to the set of official grade levels. It is represented internally as a function on the official step number set of V (the integer interval [0,256)), obtained by letting: (1) g(n) be the official level corresponding f(n), if f(n) is defined, and (2) g(n) be zero, if f(n) is undefined (Figure 2.8).

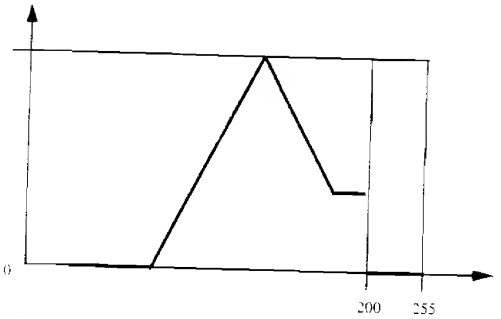


Figure 2.8 *The official membership function corresponding to the membership function shown in Figure 2.6*

Operations on official membership function

In a fuzzy inference unit, the following operations on membership functions are available:

(1) Point-by-point operations. Let O be any of the operations of grade values defined in Table 2.3. O can also be regarded as an operation on official membership functions f and g, such that h = O(f,g) if and only if h(n) = O(f(n), g(n)), for all n in [0,256).

(2) Cut operation. Let c be a grade and f be an official membership function. g = Cut(f,c) if and only if g(n) = min(f(n),c) for all n in [0,256) (Figure. 2.9).

(3) Shift operation. Let d be any integer in (-256,256) and f be an official membership function. Shift the function along the interval [0,256) a distance d (a positive distance indicates a right shift while a negative distance indicates a left shift). The empty position after shifting is filled in with the value of the position prior to shifting (Figure. 2.9).

(4) Concatenation. Let c be any integer in [0,256), and f and g are official membership functions. h = Conct(f,g,c) if and only if h(n)=f(n) for all n<c and h(n)=g(n) for all n >= c (Figure. 2.9).

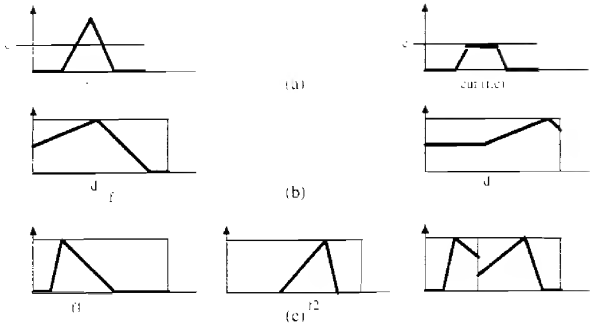


Figure 2.9 ***Operations of official membership functions: (a) cut operation, (b) shift operation, (c) concatenation***

Chapter 3 FIU Grammar

This chapter defines the syntax and semantics of an FIU program.

The terminology for the use of identifiers is summarized in Appendix B.

The general rule for numbers is the following:

There is a domain defined for each syntactic position where a number occurs. The domain is a set, and the number must belong to the domain.

The notation used in syntax definitions to follow have the following meanings:

notation	meaning
=::	define
|	or
∈	empty

Program

An independent FIU source code is called a program. A program specifies a fuzzy inference unit.

Syntax:

```
program =::
        clauseSequence endSymbol          |
        header semicolonSymbol clauseSequence endSymbol
clauseSequence =::
        varClauseSequence semicolonSymbol ruleClauseSequence
varClauseSequence =::
        varClause                         |
        varClause semicolonSymbol varClauseSequence
varClause =::
        invarClause                       |
        outvarClause
```

ruleClauseSequence =::
ruleClause |
ruleClause semicolonSymbol ruleClauseSequence

A program consists of an optional header, a sequence of varClauses and a sequence of ruleClauses. The header (if there is one) and each clause in clause sequences are followed by a semicolon. The last clause in the program is followed by an endSymbol. A varClause is either an invarClause or an outvarClause (Figure. 3.1).

Example:

fiu: invar x: outvar y: if x then y.

Semantics:

The program defines a fuzzy inference unit in the following ways:

(1) The attributes are defined by the header. If the program has no header, the default attributes are used.

(2) The variable nodes, the label nodes and the arrows between them are defined by the sequence of varClauses. Also defined by the varClause is the official membership function list.

(3) The rule nodes and their incoming and outgoing arrows are defined by the ruleClauses.

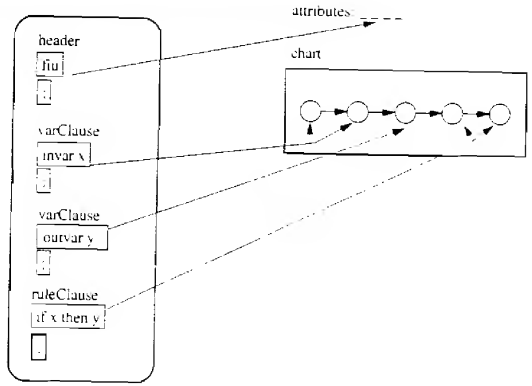


Figure 3.1 *FIU Program Defining a Fuzzy Inference Unit*

Header

A header defines the attributes of a fuzzy inference unit.

Syntax:

```
header =::
        fiuSymbol methodSpec operatorSpec gradeSpec

methodSpec =::
        tvfiSymbol      |
        mamdaniSymbol   |
        ∈
operatorSpec =::
        leftParenthesisSymbol operator operation rightParenthesisSymbol
operator =::
        minSymbol       |
        maxSymbol       |
        prodSymbol      |
        sumSymbol       |
        bunionSymbol    |
        binterSymbol
gradeSpec =::
        gradePrefix gradeFactor
grdePrefix =::
        timesSymbol     |
        ∈
gradeFactor =::
        number  |
        ∈
```

A header of a program consists of a fiuSymbol followed by an optional methodSpec, an optional operatorSpec and an optional gradeSpec. The fiuSymbol is the word "fiu". The methodSpec is either the word "tvfi" or "mamdani". The operatorSpec is a pair of operators enclosed in a pair of parentheses. The gradeSpec consists of an optional timesSymbol and an optional number, which is an integer between 4 and 16 (Figure. 3.2).

Example:

```
fiu
fiu tvfi (min max) *8
```

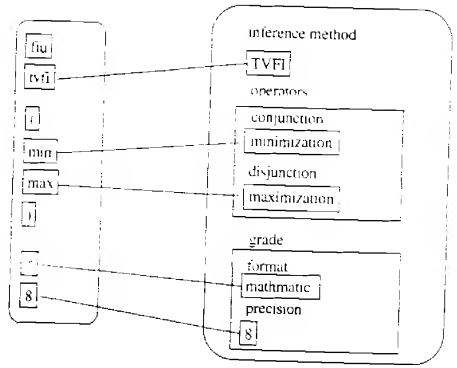


Figure 3.2 *Header Defining Attributes*

Semantics

The methodSpec defines the inference method of the fuzzy inference chart, which is either the Truth Value Flow Inference method or Mamdani's method. The methodSpec is reserved word "tvfi" or "mamdani" respectively. If there is no methodSpec in the header, the default inference method (TVFI method) is used.

The two operators in the operatorSpec defines the logical conjunction operator and the logical disjunction operator of the chart. The following is a table of operators:

Table 3.1 operators, symbols and its semantics

name	symbol	meaning
min	&	minimization operator
max	|	maximization operator
prod	*	probability product operator
sum	+	probability sum operator
bunion		boundary union operator
binter		boundary intersection operator

If there is no operatorSpec in the header, the default logical conjunction and disjunction operators (minimization and maximization operators, respectively) are used.

The gradeSpec defines the representation of grade. If there is a timesSymbol in the gradeSpec, the external form is the mathematic form; otherwise, the external form is the default level form. If there is a number in the gradeSpec, it is the grade precision; otherwise, the grade precision is the default 8.

In the subsequent source code, grades are represented in their external form.

varClause

A varClause is either an invarClause or an outvarClause. An invarClause defines an input variable node together with its outgoing arrows and the input label nodes which are the destination nodes of the arrows, and attaches attributes to the variable node. An outvarClause defines an output variable node together with its incoming arrows and the output label nodes which are the source nodes of the arrows, and attaches attributes to the variable node (Figure. 3.3). In addition, a varClause may defines several membership functions in the official membership function list.

Syntax:

```
invarClause =::
        invarSymbol invarSequence
invarSequence =::
        invarSpec          |
        invarSpec commaSymbol invarSequence
invarSpec =::
        identifier         |
        identifier engUnitSpec rangeSpec inLabList
inLabList =::
        mbfList

outvarClause =::
        outvarSymbol outvarSequence
outvarSequence =::
        outvarSpec         |
        outvarSpec commaSymbol outvarSequence
outvarSpec =::
        identifier         |
        identifier engUnitSpec rangeSpec dfzSpec outLabList
outLabList =::
        mbfList |
        singletonList
```

An invarClause is an invarSymbol followed by a sequence of invarSpecs separated by commas. An invarSpec is an identifier possibly followed by an engUnitSpec, a rangeSpec and an inLabList, which is a mbfList. An outvarClause is an outvarSymbol followed by a sequence of outvarSpecs separated by commas. An outvarSpec is an identifier possibly followed by an engUnitSpec, a rangeSpec, a dfzSpec and an outLabList. The outLabList is either a mbfList or a singletonList. If the inference method of the fuzzy inference unit is TVFI method, the outLabList must be a singletonList.

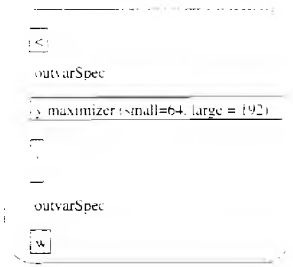


Figure 3.3 (a)

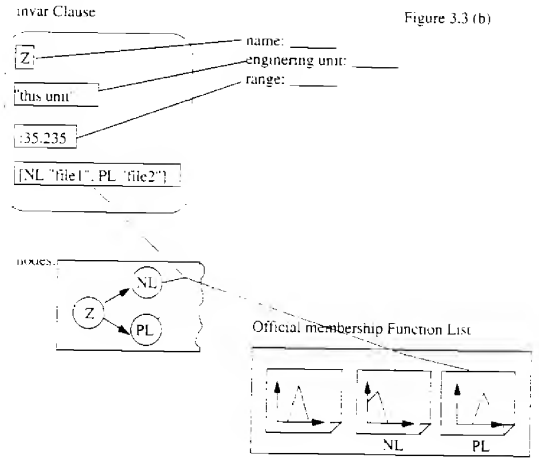


Figure 3.3 ***(a) An outvarClause consisting of outvarSpecs (b) An (invarSpec) defining an (input) variable***

The occurrence of the identifier in an invarSpec or an outvarSpec is declarational. The scope of that identifier is entire program.

Example

```
invar x
invar x1, x2
invar z "this unit" : 35, 235 [NL "file1", PL "file2]
outvar y maximizer (small = 64, large = 192), w
```

Semantics

An invarSpec defines an input variable. If the invarSpec does not have an inLabList, the node is a G-type variable, and an input label of G-type and an arrow linking the variable to the label are also defined automatically. Otherwise, the inLabList defines a series of input labels, arrows linking the variable to each of the labels, and possibly several official membership functions.

An outvarSpec defines an output variable. If the outvarSpec does not have an outLabList, the node is a G-type variable, and an output label of G-type and an arrow linking the label to the variable are also defined automatically. Otherwise, the outLabList defines a series of output labels, arrows linking the labels to the variable, and possibly several official membership functions.

The engUnitSpec, the rangeSpec and the dfzSpec in an invarClause or an outvarClause specify the engineering unit, the range and the defuzzifier of the defined variable node.

engUnitSpec

A engUnitSpec specifies the engineering unit of a variable node.

Syntax:

engUnitSpec =:: string

An engUnitSpec is a string.

Example

```
"mph"
"miles per hour"
```

Semantics

An engUnitSpec defines the engineering unit of the variable.

An invarSpec or an outvarSpec defining a G-type variable should not have an engUnitSpec.

If an invarSpec or an outvarSpec has no engUnitSpec, the default is used.

rangeSpec

A rangeSpec specifies a range.

Syntax:

```
rangeSpec =::
        colonSymbol rangeLeft stepSpec rangeRight
        |
stepSpec =::
```

```
        commaSymbol    |
        leftParenthesisSymbol rightParenthesisSymbol        |
        leftParenthesisSymbol number rightParenthesisSymbol
rangeLeft ::=
        float
rangeRight ::=
        float     |
        float minusSymbol
```

A rangeSpec is a colon followed by two numbers, the rangeLeft and the rangeRight, separated by either a comma or a pair of parentheses with or without an enclosed number, and possibly suffixed by a minus symbol. The rangeLeft should be less than the rangeRight. The domain of the enclosed number should be greater than zero but less than the difference between the rangeRight and rangeLeft.

Example:

```
:20,250
:-0.02(0.005)0.9742
:0()0.48
:0()0.48-
```

Semantics:

The two floats define the minimum and the maximum of the range. The stepSpec defines the step length: (1) if the stepSpec is a comma, the step length is 1; (2) if the stepSpec is a pair of parentheses with an enclosed positive float, the step length is equal to the positive float; (3) if the stepSpec is a pair of parentheses, the step length is calculated by dividing the interval between the minimum and the maximum into 255 subintervals with the same length (Figure 3.4).

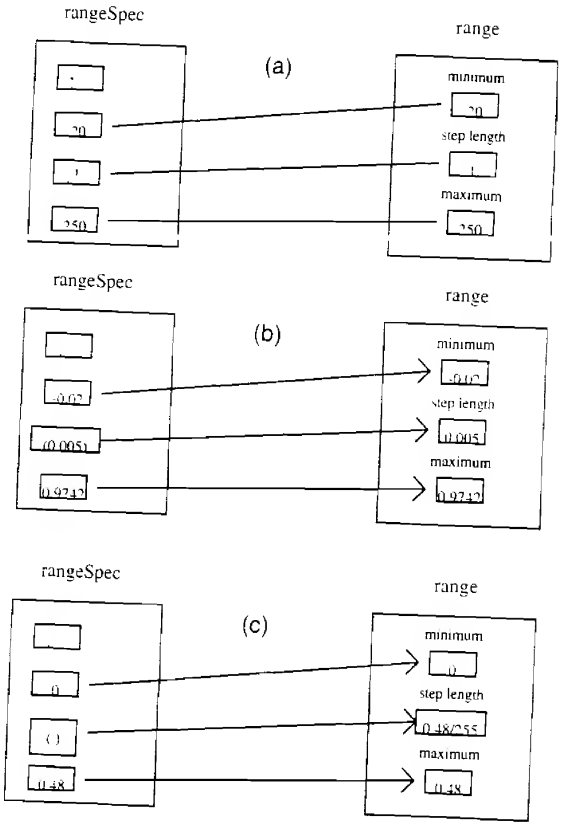


Figure. 3.4 *(a) (b) (c) rangeSpec defining ranges*

However, if the rangeSpec is suffixed by a minusSymbol, the interval is considered as a right-open interval (i.e. the maximum is not included in the range).

An invarSpec or an outvarSpec defining a G-type variable should not have a rangeSpec.

If the rangeSpec in an invarSpec or an outvarSpec defining a V-type variable is absent, the default range will be used.

dfzSpec

A dfzSpec specifies a defuzzifier.

Syntax:

dfzSpec ::=
centroidSymbol |
maximizerSymbol |
maximizerSymbol leftSymbol |
maximizerSymbol rightSymbol

The possible dfzSpecs are shown in the examples.

Example:

```
*
^
^-
^+
```

Semantics:

The defuzzifier attributes specified by the defuzzifiers are listed below.

Table 3.2 dfzSpec and corresponding defuzzifier

dfzSpec	defuzzifier
*	centroid
^	maximizer
^-	left-most maximizer
^+	right-most maximizer

mbfList and singletonList

An mbfList or a singletonList in an invarSpec or outvarSpec defines the label nodes linked to the variable defined by the invarSpec or outvarSpec, and, possibly, several official membership functions.

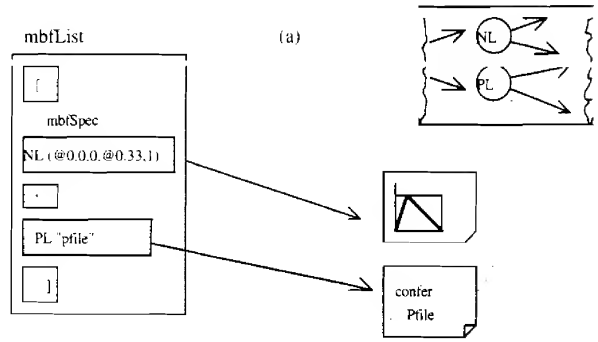
Syntax:

```
mbfList =::
        leftBracketSymbol mbfSequence BraketSymbol
mbfSequence =::
        mbfTerm                 |
        mbfTerm commaSymbol mbfSequence
mbfTerm =::
        identifier mbfSpec

singletonList =::
        leftParenthesisSymbol singletonSequence rightParenthesisSymbol
singletonSequence =::
        singletonTerm           |
        singletonTerm commaSymbol singletonSequence

mbfSpec =::
        listSpec                |
        operSpec                |
        fileSpec                |
        sameSpec                |
        copySpec                |
        shiftSpec               |
        cutSpec                 |
        concSpec
```


An mbfList is a sequence of mbfTerms enclosed in a pair of brackets and separated by commas. An mbfTerm is an identifier followed by an mbfSpec. Similarly, a singletonList is a sequence of singletonTerms enclosed in a pair of parentheses and separated by commas. A singletonTerm is an identifier followed by a singletonSpec (Figure 3.5). The mbfSpec is one of the following: listSpec, operSpec, fileSpec, sameSpec, copySpec, shiftSpec, or concSpec.



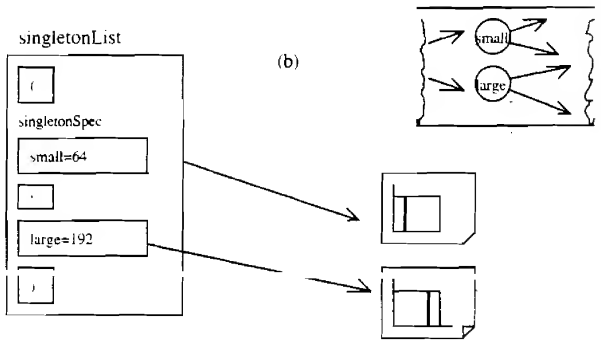


Figure 3.5 *(a) mbfList defining labels (b) singletonList defining labels*

The occurrence of an identifier in a mbfTerm or a singletonTerm is declarational. The scope of the identifier consists of (let V be the variable defined by the invarSpec or outvarSpec, of which the mbfTerm or singletonTerm is a component):

(1) The mbfList or singletonList, except the labReferences in it (see below);

(2) All labReferences and assertions, which are bounded to the variable V.

Example

```
[NL (@0.0, 0, @0.33, 1), PL "plfile"]
(small = 64, large = 192)
```

Semantics:

A mbfTerm or a singletonTerm defines a label. The identifier in the mbfTerm or the singletonTerm is the name of the label node. If the mbfSpec or the

singletonSpec is not an operSpec, it may eventually define a membership function, and that is the membership function of the label node. If a mbfTerm or a singletonTerm is not a sameSpec, it defines an official membership function in the official membership function list and assigns the function to the label. A sameSpec assigns an existing official membership function to the label.

listSpec and fileSpec

A membership function can be defined by listing function values.

Syntax:

listSpec =::
leftParenthesisSymbol termSequence rightParenthesisSymbol
termSequence =::
term |
term commaSymbol termSequence
term =::
number |
atSymbol number

fileSpec =::
string

A listSpec is a table consisting of numbers included within a pair of parentheses. The numbers in the table are separated by commas. Some numbers may be prefixed by an atSymbol. A term with an atSymbol is said to be a prefixed term. A term without an atSymbol is said to be a pure term.

The numbers in the prefixed terms must be values of the variable. The numbers in the pure terms must be grades.

The fileSpec is a string literal.

Example:

(@0.33, 0, @0.5, 1, @0.67, 0)
"pfile"

Semantics:

The listSpec defines an official membership function. The fileSpec specifies a name of a file, from which a listSpec can be read (Figure 3.6).

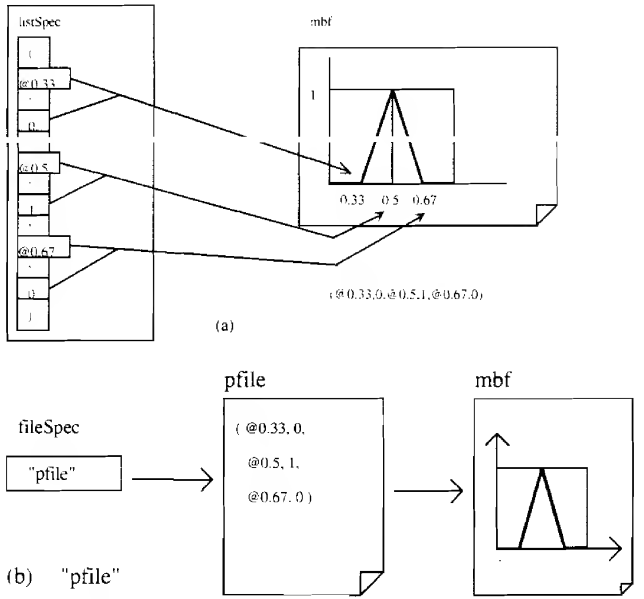


Figure. 3.6 ***(a) listSpec defining membership function, (b) fileSpec defining membership function***

The function f defined by a listSpec is obtained by enumerating the terms in the listSpec, and interpreting them in the following ways:

Enumeration:

(1) If the first term in the listSpec is a pure term, it is f(0). Subsequent pure terms are the subsequent function values f(1), f(2), ...

(2)A prefixed term is interpreted as the corresponding official step number, say n, and the pure term next to prefixed term is f(n). Subsequent pure terms are the function values f(n+1), f(n+2), ...

If, for some n, f(n) is defined more than once, the earlier definition is replaced by the later. If t(n) is not defined in the above process, leave it blank.

Interpolation:

(1) If f(0) is blank, let f(0)=0. Similarly, if f(255) is blank, let f(255)=0.

(2) If f(n) is blank, Let a be the largest number such that f(a) is defined and a<n, and b be the smallest number such that f(b) is defined and n<b. Define f(n) by linear interpolation over the interval [a,b].

Appendix C presents a numeric example of the enumeration and interpolation steps.

If there are no successive pure terms in the listSpec, the above enumeration process applies also to the (non-official) range step values to obtain a (non-official) membership function. Otherwise, the listSpec does not define a membership function.

operSpec, shiftSpec, cutSpec and concSpec

An operSpec, a shiftSpec, a cutSpec or a concSpec defines an official membership function by operations on existing membership function(s).

Syntax:

operSpec =::
operator leftParenthesisSymbol labName commaSymbol labName rightParenthesisSymbol
labName =::
identifier
identifier ofSymbol identifier
shiftSpec =::
shiftSymbol leftParenthesisSymbol labName commaSymbol float rightParenthesisSymbol
cutSpec =::
cutSymbol leftParenthesis labName commaSymbol float rightParenthesisSymbol
concSpec =::
concSymbol leftParenthesisSymbol labName commaSymbol labName commaSymbol float rightParenthesisSymbol

An operSpec, a shiftSpec, a cutSpec or a concSpec starts by specifying an operation followed by a listing of the operands. The operation is specified by an operator, a shiftSymbol, a cutSymbol, or a concSymbol. The operands are labReferences and/or numbers, separated by commas and enclosed in a pair of parentheses.

A labReference is an identifier followed by an ofSymbol and another identifier. The second identifier is an applicational (as opposed to declarational) occurrence of a variable name, identifying a V-type variable. The labReference is said to be bound to that variable. The first identifier is an applicational occurrence identifying a V-type label linked to the variable. The labReference then refers to the label.

However, if the second identifier is identical to the variable name specified by the invarSpec or outvarSpec, in which the labReference occurs, the identifier can be omitted together with the ofSymbol.

The operands of an operSpec are two labReferences. The operands of shiftSpec are a labReference and a number, where the number is a distance value of the variable specified by the invarSpec. The operands of a cutSpec are a labName and a number which is an official grade level.. The operands of a concSpec are two labNames and a step value of the variable specified by the invarSpec.

Example:

```
min( OK of x1, OK of x2)
cut( PL of theta, 0.75)
shift( label1 , -200)
conc( small of x1, small of x2, 16)
```

Semantics:

The official membership function defined by an operSpec is the result of an operation on the two official membership functions assigned to the two labels which are refereed to by the two operands of the operSpec.

The official membership function defined by shiftSpec, cutSpec and consSpec is the result of a shift operation, cut operation and concatenation operation, respectively, on the official membership function(s) assigned to the label to which the labReference operands refer. The operation is performed using the numeric parameter in shiftSpec, cutSpec and concSpec, respectively.

sameSpec and copySpec

A sameSpec or a copySpec defines a membership function which is identical to another membership function.

Syntax:

```
sameSpec =::
        sameSymbol labName

copySpec =::
        copySymbol labName
```

Example:

```
= PL of x
copy PL of x
```

Semantics:

The sameSpec or copySpec specifies a membership function identical to the membership function specified by the labReference.

In addition, a copySpec defines an official membership function in the official membership function list for the label, which is identical to the one defined by the label to which labReference refers(Figure 3.7).

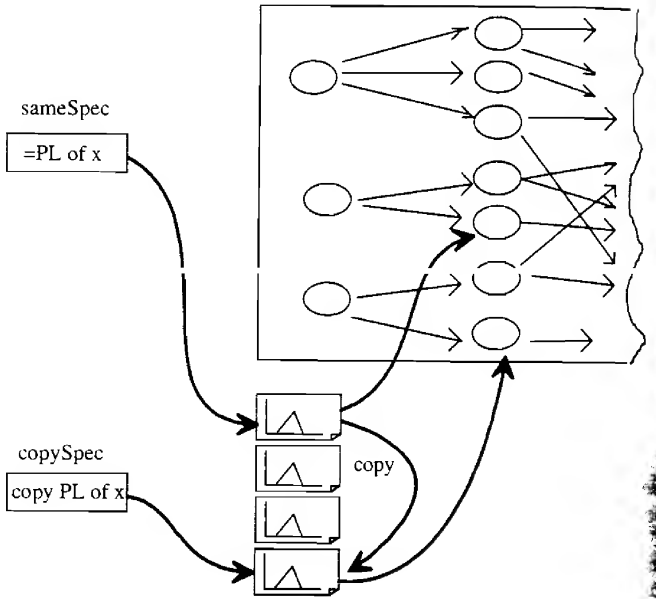


Figure. 3.7 *sameSpec and copySpec defining official membership functions*

singletonSpec

A singletonSpec specifies a singleton.

Syntax:

singletonSpec =::
isSymbol number

A singletonSpec is an isSymbol followed by a float. The float must be consistent with the range of the variable defined by the invarSpec or outvarSpec, in which the singletonSpec occurs.

Example:

= 25
is 33.3

Semantics:

A singletonSpec specifies a singleton with the float in the singletonSpec as the support point of the singleton.

RuleClause

A ruleClause defines a rule (node) together with its outgoing and incoming arrows.

Syntax:

```
ruleClause =::
        ifSymbol condition thenSymbol consequence
condition =::
        assertionSequence
consequence =::
        assertionSequence
assertionSequence =::
        assertion                                    |
        assertion andSymbol assertionSequence
assertion =::
        identifier                                   |
        identifier isSymbol identifier
```

A ruleClause consists of an ifSymbol, a condition, a thenSymbol and a consequence, where a condition or a consequence is a sequence of assertions separated by the andSymbol.

An assertion is either a single identifier or an identifier followed by an isSymbol and another identifier. If an assertion is a single identifier, it is an applicational occurrence of a G-type variable name. Otherwise, if an assertion is an identifier followed by (an isSymbol and) another identifier, the first variable is an application of a V-type variable name, and the assertion is bound to the variable. The other identifier in the assertion is an applicational occurrence of a label name (Figure 3.8).

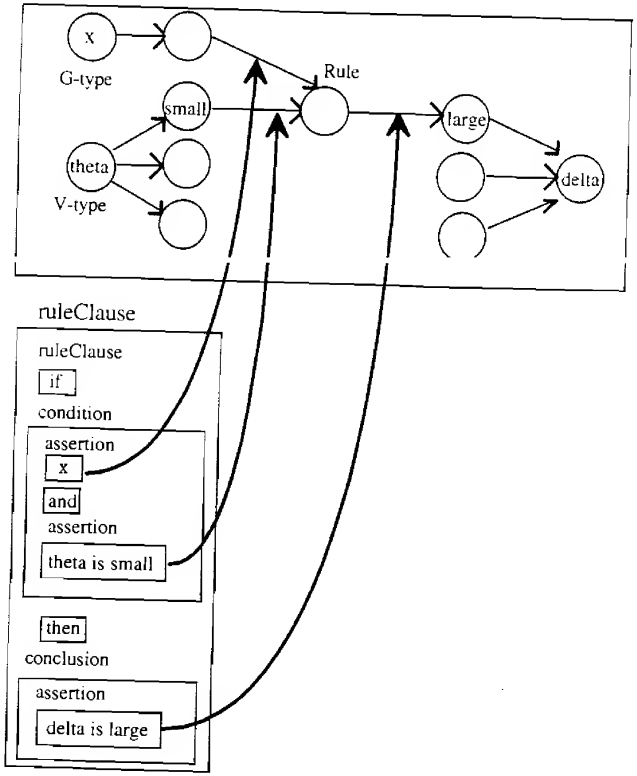


Figure 3.8 *A ruleClause defining a rule*

Example:

if theta is large then delta is small
if x and theta is small then delta is large

Semantics:

An assertion in a ruleClause specifies a label node. If the assertion is of G-type, the G-type label node linked to the G-type variable node identified by the G-type variable name in the assertion is specified. Otherwise, if the assertion is of V-type, it specifies the label node identified by the label name for the variable node identified by the V-type variable name in the assertion.

A ruleClause defines a rule node. The incoming arrows of the rule node are those from the input label nodes specified by the assertions of the ruleClause, and the outgoing arrows of the rule node are those to the output label nodes specified by the assertions.

Chapter 4 FOU Grammar

A FOU provides simple operations such as arithmetic calculation and function table reading.

The notation used in syntax definitions to follow have the following meanings:

notation	meaning
=::	define
|	or
∈	empty

Syntax:

```
program =::
        fouSymbol semicolonSymbol varList semicolonSymbol
                operationClause endSymbol
varList =::
        var     |
        var semicolonSymbol varList
var =::
        invarSymbol identifierSequence    |
        outvarSymbol identifierSequence
identifierSequence =::
        identifier          |
        identifier commaSymbol identifierSequence
operationClause =::
        calcClauseSequence |
        tableClause
```

A FOU consists of a fouSymbol, a varList and an operationClause separated by semicolonSymbols and terminated by an endSymbol.

The varList consists of a series of varSpecs separated by semicolonSymbols. A varSequence is a list of identifiers separated by commaSymbols and prefixed by either an invarSymbol or an outvarSymbol (Figure. 4.1).

The operationClause is either a calcClauseSequence or a tableClause. A program with a tableClause must have exactly one output variable.

Example:

```
fou;
invar a;
invar b;
invar c;
outvar u;
calculate u=(a-0.583)*b/(c+a*a)-4 .
```

Semantics:

The identifiers in the varList specifies variables. A varSpec prefixed by an invarSymbol specifies input variables, while a varSpec prefixed by an outvarSymbol specifies output variables.

The operationClause specifies operations of the FOU.

Note: If the operationClause ends with a number and the endSymbol occurs as a period ("."), a blank space is needed between the number and the period.

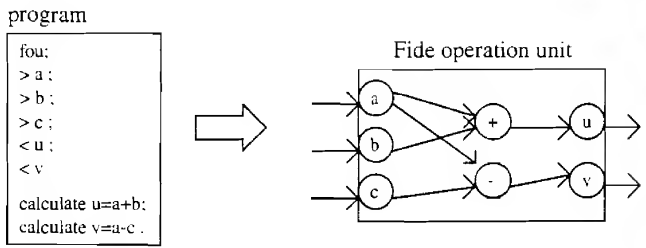


Figure 4.1 ***FOU program defining operations***

calcClauseSequence

The calcClauseSequence specifies arithmetic operations.

Syntax:

```
calcClauseSequence =::
        calcClause          |
        calcClause semicolonSymbol calcClauseSequence
calcClause =::
        calcSymbol identifier isSymbol expression
expression =::
        term      |
        term plusSymbol expression          |
        term minusSymbol expression
term =::
        factor
        factor timesSymbol factor |
        factor divideSymbol factor
factor =::
        float      |
        identifier |
        minusSymbol identifier |
        leftParenthesisSymbol expression rightParenthesisSymbol |
        identifier factor
```

The calcClauseSequence is a sequence of calcClauses separated by semicolonSymbols. A calcClause is a sequence of a calcSymbol, an identifier, an isSymbol, and an expression, in that order. The identifier is an output variable. The expression is an ordinary arithmetic expression consisting of numbers, input variables, addition, subtraction, multiplication, division, elementary functions and parentheses. The following is a table of the elementary functions supported by the Fide Compiler:

Table 4.1 Elementary functions

Name	Description
abs	Returns the absolute value of its argument.
sqrt	Computes the square root of an positive floating point number.
exp	Computes the exponential of a floating point number.
pow10	Computes the value of 10 raised to the power of the argumenmt of pow10.
log	Computes the natural logarithm of a positive floating point number.
log10	Computes the base-10 logarithm of a positive float point number.
sin	Computes the sine of an angle in radians.
cos	Computes the cosine of an angle in radians.
tan	Computes the tangent of an angle in radians.
asin	Computes the arc sine of a value between -1 and 1 and returns an angle between $-\pi/2$ and $\pi/2$ in radians.
acos	Computes the arc cosine of a value between -1 and 1 and returns an angle between and π in radians.
atan	Computes the arc tangent of a value and returns an angle between $-\pi/2$ and $\pi/2$ in radians.
sinh	Computes the hyperbolic sine of its argument.
cosh	Computes the hyperbolic cosine of its argument.
tanh	Computes the hyperbolic tangent of its argument.

Note: (1) The timeSymbol can not be omitted. For example, 2*x can not be abbreviated to 2x or 2 x. (2) If the argument of a function is an expression, the

expression should be enclosed in a pair of parenthesis. For example sin(x*y) is different from sin x *y, which is equal to y*sin x. Multiple nestings are allowed. For example, exp sin x is equal to exp(sin x).

Example:

calculate u=(a-0.583)*b/(c+d*d)-4; calculate v=(a-c)*atan(b/d)

Semantics:

A calcClause specifies the arithmetic operations needed to compute the value of an output variable by evaluating the expression.

tableClause

A tableClause specifies a function table and the way to read function values from the table.

Syntax:

```
tableCluase ::=
        tableSymbol tableHeader table
tableHeader ::=
        leftParenthesisSymbol tableHederItemList rightParenthesisSymbol
tableHeaderItemList ::=
        tableHeaderItem  |
        tableHeader Item commaSymbol tableHeaderItemList
tableHeaderItem ::=
        identifier colonSymbol number
table ::=
        leftParenthesisSymbol tableItemList rightParenthesisSymbol |
        leftParenthesisSymbol identifier tableItemList rightParenthesisSymbol
tableItemList =::
        tableItem |
        tableItem commaSymbol tableItemList
tableItem ::=
        tableTerm  |
        atSymbol number commaSymbol tableTerm
tableTerm ::=
        number |
        table
```

A tableClause consists of a tableSymbol, a tableHeader and a table, in that order.

A tableHeader is a series of tableHeaderItems separated by commas and enclosed in a pair of parentheses, where a tableHeaderItem is an identifier and a number, separated by a colonSymbol.

Each of the input variables must occur as an identifiers in the tableHeader exactly once. Also the number in a tableHeaderItem must be a positive integer, which is said to be the item number of the corresponding input variable. In the current implementation, the product of item numbers in the tableHeader is limited to 32767.

A table is a series of tableItems separated by commas, prefixed by an optional identifier, and enclosed in a pair of parentheses. A tableItem is a number of a table, possibly prefixed by an atSymbol followed by a number and a comma.

The outer-most table, which is an immediate component of the tableClause, is said to be at level 1. It is the only table at level 1. However, if there are more then one input variable, all of the tableTerms in the table at level one must be tables, which are said to be tables at level 2. In general, the maximal level is equal to the number of input variables. Only the tableItems in a table at this level are numbers. A table at a smaller level n has as its tableItems tables at the level n+1.

A table at level k corresponds the k-th input variable in the tabelHeader. The identifier occurring as the immediate component of the table, if any, must be the same as the corresponding input variable. The number of tableItems in a table must also be equal to the item number of the corresponding input variable. If a tabelItem in the table has a prefix, the number in the prefix must be equal to the ordinal number of the tableItem in the table.

Note: The identifiers and the number prefixing tableItems are redundant. They are used for documentation purposes and as cross checking aids, which become significant for larger tables.

Example:

```
table (x : 3, y: 3, z: 5)
(x @0,  (y @0,  (z 15, 16, 19, 24, 31),
          @1,  (z 15, 16, 19, 24, 31),
          @2,  (z 15, 16, 19, 24, 31) ),
 @1,  (y @0,  (z 15, 16, 19, 24, 31),
          @1,  (z 16, 17, 20, 25, 32),
```

```
        @2,  (z 17, 18, 21, 26, 33) ),
@2,  (y @0,  (z 15, 16, 19, 24, 31),
        @1,  (z 17, 18, 21, 26, 33),
        @2,  (z 19, 20, 23, 28, 35) ) ).
```

Semantics:

The input variable values must be positive integers less than the number of the corresponding variables. The output value of the (only) output variable is a number in the table. Select the tableItem in the table at level one, according to the value of the first input variable. If the selected tableItem is a table (at a higher level), select the tableItem in its table. Repeat the selection process until the selected tableItem is a number, which is the desired output value. The input variable values can be thought of as indexes of a k dimensional array, and the output values are the values of the array.

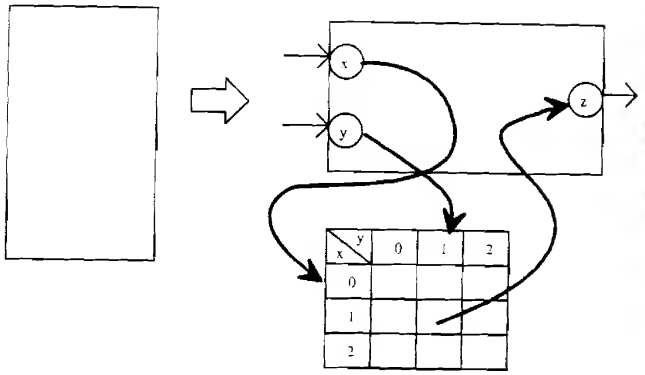


Figure 4.2 ***tableItem specifying a function table (2 dimensional example)***

Chapter 5 FEU Grammar

A FEU provides an simple interface for Fide Composer to access existing modules developed in C.

The notation used in syntax definitions to follow have the following meanings:

notation	meaning
=::	define
|	or
∈	empty

Syntax:

```
program =::
        header semicolonSymbol varList semicolonSymbol callClause
        endSymbol
header =:: feuSymbol string
varList =::
        varSpec                 |
        varSpec semicolonSymbol varList
varSpec =::
        invarSymbol identifierSequence
identifierSequence =::
        identifier              |
        identifier commaSymbol identifierSequence
callClause =:: callSymbol identifier
```

An FEU consists of a header, a varList and a callClause, separated by semicolonSymbols and terminated by an endSymbol.

The header is a feuSymbol followed by a string.

The varList consists of a series of varSpecs separated by semicolonSymbols. A varSpec is a list of identifiers separated by commaSymbols and prefixed by either an invarSymbol or an outvarSymbol.

The callClause is a callSymbol followed by an identifier.

Example:

```
feu "TRUCK";
> gear_position;
> throttle;
< speed, distance;
call simulate;
```

Semantics:

(1) The string in the header specifies a legal DOS file name, say xxxx, for which a file with the DOS file name "xxxx.obj" exists. The file must be valid in the sense that it has compiled successfully using the Turbo C compiler (version 2.0 and up).

(2) The identifiers in the varList specifies variables. A varSpec prefixed by an invarSymbol specifies input variables, while a varSpec prefixed by an outvarSymbol specifies output variables.

Interface with C Program:

The identifier in the callClause, say "ff", should be a legal ANSI C function name, and the file "xxxx.obj" should implement the following functions:

```
ff
init_ff
close_ff
```

Moreover, the function ff must have a parameter list of "float *" type parameters and return zero for exception or non-zero for success. The type of the functions "init_ff" and "close_ff" should be "void(void)" (Figure. 5.1).

Note: Conceptually, Fide composer makes a C program, in which the function specified in the callClause is called in the form:

```
ff ( &v1, ... , &vn)
```


where ff stands for the function name, and v1, ... , vn stand for the variable names (the identifiers in the varSpecs). However, Because all variables will be renamed, the variable names play no role in the semantics.

The order of variables in the above function call is assumed to have been arranged in the object file "xxxx.obj" according to the following principles: (a) all variables are segregated as input or output variables with input variables occurring before output variables, and (b) the order within input variables and output variables are maintained from the original source statements.

Although not necessary, it is good to specify all input variables before output variables in a FEU program.

Example:

In the header file TRUCK.H of the truck simulating program, we need the following functions:

```
void  init_simulate (void);
        /* initial graphics, set initial values, and so on */

void  close_aimulate(void);
        /* close the simulation,  release memory, close graphics, ...*/

int  simulate(float *gear_position, float *throttle, float *speed, float
*distance);
        /* simulate the truck's behavior    */
        /* returns zero for exceptions      */
```

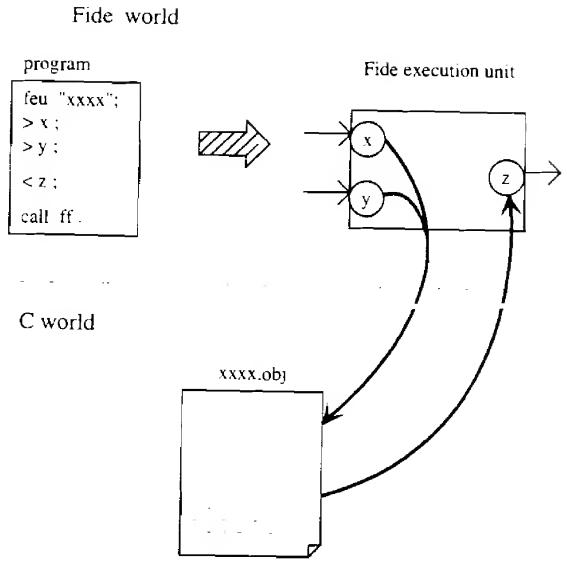


Figure. 5.1 ***FEU defining an interface to existing C programs***

Appendix A Context Free Grammar

The context-free grammar of FIL is given in syntactic formulas. The formulas are divided into 3 groups: the formulas for FIU, the formulas for FOU and the formulas for FEU.

FIU

```
program =::
        clauseSequence endSymbol          |
        header semicolonSymbol clauseSequence endSymbol
clauseSequence =::
        varClauseSequence          |
        varClauseSequence semicolonSymbol ruleClauseSequence
varClauseSequence =::
        varClause          |
        varClause semicolonSymbol varClauseSequence
varClause =::
        invarClause        |
        outvarClause
ruleClauseSequence =::
        ruleClause         |
        ruleClause semicolonSymbol ruleClauseSequence

header =::
        fiuSymbol methodSpec operatorSpec gradeSpec
methodSpec =::
        tvfiSymbol         |
        mamdaniSymbol      |

operatorSpec =::
        leftParenthesisSymbol operator operation
                rightParenthesisSymbol
operator =::
        minSymbol          |
        maxSymbol          |
        prodSymbol         |
        sumSymbol
```

```
        bunionSymbol
        binterSymbol
gradeSpec =::
        gradePrefix gradeFactor
grdePrefix =::
        timesSymbol     |
        ∈
gradeFactor =::
        integer         |
        ∈

invarClause =::
        invarSymbol invarSequence
invarSequence =::
        invarSpec       |
        invarSpec commaSymbol invarSequence
invarSpec =::
        identifier      |
        identifier engUnit range inLabList
inLabList =::
        mbfList

outvarClause =::
        outvarSymbol outvarSequence
outVarSequence =::
        outvarSpec      |
        outvarSpec commaSymbol outvarSequence
outvarSpec =::
        identifier      |
        identifier engUnitSpec rangeSpec dfzSpec outLabList
outLabList =::
        mbfList |
        singletonList

engUnitSpec =:: string
rangeSpec =::
        colonSymbol rangeSpec   |
```



```
        ∈
rangeSpec =::
        rangeLeft precision rangeRight
precision =::
        commaSymbol     |
        leftParenthesisSymbol rightParenthesisSymbol        |
        leftParenthesisSymbol positive float rightParenthesisSymbol
rangeLeft =::
        float
rangeRight =::
        float   |
        float minusSymbol

dfzSpec =::
        centroidSymbol  |
        maximizerSymbol |
        maximizerSymbol leftSymbol |
        maximizerSymbol rightSymbol

mbfList =::
        leftBracketSymbol mbfSequence BraketSymbol
mbfSequence =::
        mbfTerm         |
        mbfTerm commaSymbol mbfSequence
mbfTerm =::
        identifier mbfSpec
mbfSpec =::
        listSpec
        operSpec |
        fileSpec |
        sameSpec |
        copySpec |
        shiftSpec |
        cutSpec |
        concSpec
singletonList =::
        leftParenthesisSymbol singletonSequence rightParenthesisSymbol
```

```
singletonSequence =::
        singletonTerm    |
        singletonTerm commaSymbol singletonSequence
singletonTerm =::
        identifier singletonSpec
singletonSpec =::
        isSymbol float

listSpec =::
        leftParenthesisSymbol termSequence rightParenthesisSymbol
termSequence =::
        term    |
        term commaSymbol termSequence
term =::
        float    |
        atSymbol float

fileSpec =::
        string

operSpec =::
        operator leftParenthesisSymbol labName commaSymbol labName
                        rightParenthesisSymbol
labName =::
        identifier          |
        identifier ofSymbol identifier

shiftSpec =::
        shiftSymbol leftParenthesisSymbol labName commaSymbol float
                rightParenthesisSymbol

cutSpec =::
        cutSymbol leftParenthesis labName commaSymbol float
                rightParenthesisSymbol
concSpec =::
        concSymbol leftParenthesisSymbol labName commaSymbol labName
                commaSymbol  float rightParenthesisSymbol
```



```
sameSpec =::
        sameSymbol labName

copySpec =::
        copySymbol labName

ruleClause =:: ifSymbol condition thenSymbol consequence
condition =:: assertions
consequence =:: assertionSequence
assertionSequence =::
        assertion          |
        assertion andSymbol assertionSequence
assertion =::
        identifier          |
        identifier isSymbol identifier
```

FOU

```
program =::
        fouSymbol semicolonSymbol varList semicolonSymbol
                operationClause endSymbol
varList =::
        var     |
        var semicolonSymbol varList
var =::
        invarSymbol identifierSequence    |
        outvarSymbol identifierSequence
identifierSequence =::
        identifier          |
        identifier commaSymbol identifierSequence
operationClause =::
        calcClauseSequence |
        tableClause

calcClauseSequence =::
        calcClause          |
        calcClause semicolonSymbol calcClauseSequence
calcClause =::
        calcSymbol identifier isSymbol expression
expression =::
        term    |
        term plusSymbol expression          |
        term minusSymbol expression
term =::
        factor
        factor timesSymbol factor |
        factor divideSymbol factor
factor =::
        float     |
        identifier |
        minusSymbol identifier |
        leftParenthesisSymbol expression rightParenthesisSymbol |
        identifier factor
```



```
tableClause ::=
        tableSymbol tableHeader table
tableHeader ::=
        leftParenthesisSymbol tableHederItemList rightParenthesisSymbol
talbeHeaderItemList ::=
        tableHeaderItem  |
        tableHeader Item commaSymbol tableHeaderItemList
tableHeaderItem ::=
        identifier semicolonSymbol number
table ::=
        leftParenthesisSymbol tableItemList rightParenthesisSymbol |
        leftParenthesisSymbol identifier tableItemList rightParenthesisSymbol
tableItemList =::
        tableItem |
        tableItem commaSymbol tableItemList
tableItem ::=
        tableTerm  |
        atSymbol number commaSymbol tableTerm
tableTerm ::=
        number |
        table
```

FEU

```
program =::
        header semicolonSymbol varList semicolonSymbol callClause
                endSymbol
header =:: feuSymbol string
varList =::
        varSpec          |
        varSpec semicolonSymbol varList
varSpec =::
        invarSymbol identifierSequence |
        invarSymbol identifierSequence
identifierSequence =::
        identifier       |
        identifier commaSymbol identifierSequence
callClause =:: callSymbol identifier
```

Appendix B Identifiers

Overview

The following are general rules relating identifiers in FIU programs.

An identifier has a scope, which is either the entire program or consist of several segments of the program. The first occurrence (in textual order) of an identifier in a scope declares a node in the fuzzy inference chart, and this occurrence is said to be the declarational occurrence of the identifier. Subsequent occurrences of the same identifier in the same scope refer to the same node and are called applicational occurrences. On the other hand, an occurrence is declarational or applicational depending on its syntactic position of occurrence. Therefore, an identifier in a scope should occur exactly once at a declarational position.

Definitions

The identifiers in a program are divided into three groups:

(1) G-type variable name
(2) V-type variable name
(3) label name

An identifier is a variable name, if its first occurrence is in an invarSpec or an outvarSpec as the identifier immediately following the invarSymbol or outvarSymbol. The occurrence is the declarative occurrence of the variable name, and the variable name is said to be declared in the invarSpec or the outvarSpec. The scope of the variable name is the entire program. Other occurrences of the name are applicational. An applicational occurrence of a variable is always in a labReference or an assertion, and the labReference or assertion is said to be bound to the variable name.

An identifier is a label name if its first occurrence is in an invarSpec or an outvarSpec as the first identifier of a mbfTerm or singletonTerm. The occurrence is the declarative occurrence of the label name. The label name is said to be associated with the variable name declared in the invarSpec or

outvarSpec. The scope of the label name consists of the following components: (1) The invarSpec or an outvarSpec, and (2) all labReferences and assertions bound to the variable name. Other occurrence of the label name are applicational.

Appendix C listSpec Example

The following is an example of listSpec, showing the enumeration and the interpolation in the semantics of listSpec. The default grade precision and official range are assumed, so that the membership function is a mapping from the integer interval [0,256) to the same interval.

Example

(@27,0,1,3,6,10,15,@64,255,@96,15,10,6,3,1)

Let f be the official membership function specified by the above listSpec.

In the enumeration step, we find the following definitions:

f(27)=0
f(28)=1
f(29)=3
f(30)=6
f(31)=10
f(32)=15
f(64)=255
f(96)=15
f(97)=10
f(98)=6
f(99)=3
f(100)=1
f(101)=0

Now we let:

f(0)=0
f(255)=0

There are four blank segments remaining: [1,26], [33,63], [65,95] and [102,255]. By using interpolation, we obtain:

```
f(n)=0                          for n in [1,26]
f(n)=15+(n-32)*240/32           for n in [33,63]
f(n)=255-(n-64)*240/32          for n in [65,95]
f(n)=0                          for n in [102,255]
```

Note: the same example can be written in different ways as shown below:

```
(@27,0,1,3,6,10,15,@96,15,10,6,3,1,@64,255)
(@25,0,0,0,1,3,6,10,15,@64,255,@96,15,10,6,3,1,@255,0)
```

An instructional exercise would be to find additional alternative forms of the example.

Appendix D Rule Macros

FIL rule macro provides an effective means to reduce the length of FIU source code as well as to increase its readability.

Syntax

FIL rule macros can occur in FIU program at any position where a ruleClause can occur.

A FIL rule macro consists of an if Symbol, a conditionMacro, a thenSymbol and a conclusionMacro. Both the conditionMacro and the conclusionMacro are sequences of assertionMacros separated by andSymbols. An assertionMacro is an identifier followed by an isSymbol and an identifierMacro. An identifierMacro is either an identifier, an underscoreSymbol, or a sequence of items separated by commaSymbols and enclosed in a pair of braces with each of the items being an identifierMacro.

The following is an example of a rule macro:

```
if
x is {L1,L2} and y is {{M1,M2,M3}}
then
z is {{_,N1,N2},{N2,N3}}
```

in which, the following are identifierArrays:

```
{L1,L2,L3}
{{M1,M2,M3}}
{{_,N1,N2},{N2,N3}}
```

Expansion

A rule macro is expanded in the following steps:

(1) Make a copy of the rule macro, removing the outer-most braces of each identifier macro. For each pair of removed braces, erase all but the first

enclosed item. Again make a copy of the original rule macro, removing the outer-most braces of each identifier macro. For each pair of the removed braces, erase all but the second enclosed item. Repeat the above procedure until the items in the removed braces are exhausted. If the items in a pair of the removed braces are exhausted before other pairs, use the last item repeatedly.

Upon completion of the above step, remove the original copy of the rule macro, and use the resulting several new rule macros.

(2) If the new rule macros do not contain braces, stop the expansion. Otherwise, if any of the obtained rule macros contain braces, perform the above step on it and replace the rule macro with further resulting ones.

(3) Repeat step (2) until the expansion is exhausted. Remove all rules containing underscoreSymbols.

Example

Consider the above example of a rule macro.

```
if
x is {L1,L2} and y is {{M1,M2,M3}}
then
z is {{_,N1,N2},{N2,N3}}
```

In step (1), we obtain the following copies or the macro:

```
if x is L1 and y is {M1,M2,M3} then z is {_,N1,N2}
if x is L2 and y is {M1,M2,M3} then z is {N2,N3}
```

note that there is only one item in the identifierMacro after y in the original rule macro and it is repeatedly used.

Perform the step (1) for the first rule macro obtained hereby, we obtain:


```
if x is L1 and y is M1 then z is _
if x is L1 and y is M2 then z is N1
if x is L1 and y is M3 then z is N2
```

similarly, from the second rule macro in the previous set, we obtain:

```
if x is L2 and y is M1 then z is N2
if x is L2 and y is M2 then z is N3
if x is L2 and y is M3 then z is N3
```

note that the identifier "N3" is repeated.

Now we have 6 rule macros, and no braces are left. However, an underscoreSymbol is found in the first among them. By removing it, we obtain five ruleClauses.

Appendix E Data File

Overview

Fide simulator reads input data from disk files, called data files. In its complete form, a data file is a matrix of values such that each column of the matrix represents values of one input variable and each row represents a set of values for all the input variables at each step. The number of columns in a data file is the number of the input variables, which must be specified at the very beginning of the data file. Usually a data file in complete form is very large, and it is difficult to construct it by hand. Fortunately, correlation among steps often exist, which makes it is possible to use abbreviations in constructing the data file. The Fide compiler, when applied to data files, converts a user data file written in the abbreviated format into a complete data file.

Format of data file

An data file consists of a head line, a number of item lines and the indicator of the end of the input data file. A line is a sequence of characters terminated by a semicolon ";", which is referred as the line delimiter. The indicator of the end of an input data file is either a period "." or the word "end". Comments are preceded by a dollar sign "$" and terminated by a carriage return. Any control characters are ignored.

The first line in an input file is the head line and the lines following the head line are item lines (any comments and control characters are ignored).

The head line

The head line consists of a sequence of input variables separated by a comma ",". Each variable in the head line can be followed by a rangeSpec which is identical to the rangeSpec in FIU.

The head line declares variables and their ranges.

Item lines

An item line consists of a sequence of items separated by a comma "," and possibly prefixed by an indicator *n , where n is a positive integer indicating how many times the item line should be read repeatedly by the compiler.

Items

An item in an item line has the following possible formats:

1) *:n*
2) *n*
3)*var:n*
4)*var n*

where *n* is a number, *var* is a variable declared in the head line.

Each item in an item line belongs to a unique variable in the head line. The variable to which an item belongs is either indicated by the variable name in the item or indicated by the position of the item in the item line. In the latter case, the i-th item corresponds to the i-th variable in the head line.

The formats have the following meanings:

1) increment the previous sample by n and take the result as current sample. The *n* in this format can be omitted and the default value of n is 0. In this case, the item become a single colon. You may omit a single colon, if you wish.

2) an immediate number is taken as the value, i.e. the variable takes *n* as its value.

3) similar to format 1). Increment var by *n* and take the result as the current sample of var. The *n* can be omitted. The default *n* is 0.

4) similar to the format 2), but the *n* can not be omitted.

An item with a *var* (format 3 and 4) should occur at the position determined in the head line. That is to say, if a *var* is the i-th variable in the head line, then it must occur as the i-th item in an item line. However, if several items left of the position is empty (or ":", ":0"), the comma between these items can be omitted as well. Therefore, the *var* may occupy a position left of the position it should occupy, and this is the case, the compiler will insert the omitted items.

Semantics

The semantics of the items lines is explained in terms of an expansion algorithm below. However, the user should not consider this algorithm as what is actually used to implement the expansion.

Step 1

Each item line starting with an indicator *n is replaced by n copies on the line with the * indicator removed.

As a result, no line with an indicator should occur in the file, and the total number of lines could (usually would) be increased.

Step 2

Scan each line from left to right until an item in format 3) or 4) is found. Then compare the position of the *var* and its position in the header line. If it is in an earlier position, insert enough items ":0" with commas to adjust the position. Then remove the *var* and the colon after it. This procedure should be repeated until no *var* occurs in the line.

As the result, all item in format 3) and 4) are replaced.

Step 3

Count the number of items in each item line. If an item line contains less item than the variables declared in the head line, append enough items ":0" (and necessary commas) at the end of the item line.

As a result, all item lines are equal in length. The data file becomes a matrix of items.

Step 4

If there is any empty item or an item ":", replace it with the item :0.

Step 5

Scan each column of the item matrix, starting from the second item line . If there is an item in format :*n*, say :*n*, replace it with the (algebraic) sum of the item in the previous line and *n* (note that *n* may be negative).

As a result, all items are plain numbers. And the matrix is the desired input data, with each of the lines corresponding to a sample, and each of the column corresponding to a sample.

Chapter 1 Preliminary

Overview

FCL, the Fide Composer Language, is a language for describing data flows (communications) among components of a fuzzy inference system. The components of a fuzzy inference system are Fide units. The Fide units currently supported are the Fide Inference Unit, the Fide Operation Unit and the Fide Execution Unit. A fuzzy inference system, described by FCL, consists of three parts. The first part is a collection of Fide units, which are given by file names of the corresponding units. The second part is a collection of system variables, which are used to communicate among the units in the system. The third part consists of a set of system specifications specifying data flow patterns (linkage relations) among the units through the system variables and variables of the units. Details on the three parts of a fuzzy inference system will be described in later sections.

Fuzzy inference systems can also be described as graphics. For how to create and edit a graphic, refer to the **Fide User's Guide** in the graphic editor section of the Composer. In the last chapter of this document we will describe the meaning of each component of a graphic and the relationship between a graphic representation and an FCL representation of a system.

The following terms are used in the ordinary sense and are not defined formally:

Identifier
An identifier is a string of letters and/or digits in which the first character is a letter. A letter is either a lower case letter (from a to z) , an uppercase letter (from A to Z) or the underscore _ character. An uppercase letter is considered different from the corresponding lowercase letter. (FCL is case sensitive.)

Number and integer
A number is zero or a string of decimal digits beginning with a non zero digit. An integer is either a number or a number prefixed by the plus sign or the minus sign.

Positivefloat and float
A positivefloat is a number or two strings of decimal digits separated by a decimal point. A float is either a positivefloat or a positivefloat prefixed by a plus sign or a minus sign.

Strliteral
A strliteral is a string of characters included in a pair of double quotation marks. (Any reserved word or symbol within a stringliteral loses its special meaning.)

Reserved words
A reserved word is a string of letters that plays a predefined syntactic role and is therefore can not to be used as an identifier. FCL has only a few reserved words. They are listed below:

Reserved Word	Meaning
UNIT	signals that the following part is a collection of units
VAR	signals that the following part is a collection of system variables
FLOAT	the following variable is of float type
LOOP	signals that the following part is a collection of data flow specifications and the specifications specify a loop system, i.e. at least one input system variable is the same as an output variable.
DO	same as LOOP except that the system is an open unlooped system.
END	signals the end of the system (description in FCL)

Besides the above reserved words, FCL uses the following special characters:

Special character	Meaning
=	assignment operator
:	operator for setting up a data flow connection
,	item delimiter for separating same kind of items
;	statement delimiter for separating statements

{ }	begin and end of linker in a call statement

Reserved words and special characters, are regarded as terminate symbols in the syntactic formulas and are used in the formulas.

Chapter 2 The Grammar and Its Description

In this chapter we first give the syntactic formulas for FCL and then describe them in English. Readers who are not familiar with syntactic formulas can refer to the English description alone.

The notation used in syntax definitions to follow have the following meanings:

notation	meaning
=::	define
|	or
∈	empty

Syntactic Formulas For FCL

```
program =:: UNIT unitlist ; VAR variablelist ; LOOP body END. |
                UNIT unitlist ; VAR variablelist ; DO body END.
unitlist =:  identifier = strliteral, unitlist |
             identifier = strliteral
variablelist =:: FLOAT identifier = float ,variablelist |
              FLOAT identifier = float
body =:: statement ; body | statement
statement =:: send | call
send =:: identifier : rightpart
rightpart =:: identifier identifier | identifier
call =:: identifier { linker }
linker =:: send , linker | send
```

Description of FCL

Program

An independent FCL source code is called a program. A program starts with the reserved word UNIT followed by a unitlist, a semicolon, the reserved word VAR followed by a variablelist, a semicolon, the reserved word LOOP or DO followed by a body, and terminated by the reserved word END. Each the

reserved words of UNIT, VAR and LOOP (or DO) begins a part of a program. The first two parts, the part starting with UNIT and the part starting with VAR, are declarations of the units in the system and the system variables, and the third part, starting with LOOP (or DO), designates the data flow patterns in the system.

Example:

```
UNIT    op1 = "C:\fide\op1",
        op2 = "C:\fide\op2",
        F1 = "C:\fide\ex4",
        op3 = "C:\fide\op3",
        op4 = "C:\fide\op4",
        F2 = "C:\fide\ex3",
        P1 = "C:\fide\pdemo";

VAR     FLOAT     x = 0.0,
        FLOAT       dx = 0.0,
        FLOAT       t = 0.0,
        FLOAT       dt = 0.0;

LOOP    x : op1 x ;
        dx : op2 x ;
        op1 { y : F1 theta };
        op2 { y : F1 omega };
        t : op3 a ;
        F1 { voltage : op3 b };
        dt : op4 x ;
        op3 { y : F2 pendulum_angle };
        op4 { y : F2 pendulum_velocity };
        F2 { control_force : P1 f };
        P1 { x : x ,
        dx : dx ,
        t : t,
        dt : dt
          }
END.
```


In the above example, the first part gives the components (Fide units) of the system. The components are named op1, op2, op3, op4, F1, F2 and P1, and specified by the files C:\fide\op1, C:\fide\op2, etc. . It is assumed these files exist and represent Fide units. The second part of the example gives the system variables x, dx, t and dt, and initializes them to 0.0 . These variables are of FLOAT type, which is the only type supported by the current version of FCL. The third part of the example gives a set of data flow specifications which describe how the units get input values and where the outputs of the units will go. For example, x : op1 x; says that the input variable x of the unit op1 gets its value from the system variable x. Another example, op1 { y : F1 theta }; says that the input variable theta of the unit F1 gets its value from the output variable y of the unit op1. In a data flow specification, we always assume that the variable on the left of : is used as output variable, which provides input value for the variable on the right of : . The variable on the right of : is an input variable which gets its value from the variable on the left of :. The semicolon : is considered as an operator setting up a communication channel from a variable of one unit (or system variable) to a variable of another unit (or a system variable). Please note, in the current implementation of FCL, the communication channel (or data flow path) must be from one unit to a different unit or between a unit and system variables. One can not set up a communication channel from a unit to itself.

Units

The units (components) of a Fuzzy inference system are given in the first part of the program describing the system. This part starts with the reserved word UNIT followed by a list of assignments separated by commas and terminated by a semicolon. Each assignment consists of three parts, the left part, the assignment operator = and the right part. The left part is a name of a unit, which will be referred to by the data flow specifications of the system. A name can be any legal identifier. The right part is a file name given in double quotation marks. A file name can be any legal DOS file name without extension or with the extension "map", and can include or omit a full path of the file. When a file name appears in a program describing a fuzzy inference system, it is the programmer's responsibility to make sure that the file exists

and represents a Fide unit (as a convention, the file should have the "map" extension) .

Example:

```
UNIT        op = "C:\fide\op",
            F  = "force.map";
```

Variables

The variables in a Fuzzy inference system are given in the second part of the program describing the system. This part starts with the reserved word VAR followed by a list of assignments separated by commas and terminated by a semicolon. Each assignment consists of three parts, the left part, the assignment operator = and the right part. The left part is a type name followed by a variable name . In the current implementation of FCL, only one type, namely FLOAT, is allowed. A variable name can be any legal identifier and it will be referred to by the data flow specifications of the system. The right part is a float number which gives an initial value to the variable in the left part.

Example:

```
VAR         FLOAT x = 0.0,
            FLOAT y = 1.5;
```

Data flow specifications

The data flow patterns in a Fuzzy inference system are given in the third part of the program describing the system. This part starts with the reserved word LOOP or the reserved word DO followed by a list of statements separated by semicolons. The reserved word LOOP indicates that the system is a closed loop, while the reserved word DO indicates that the system is an open loop, i.e. the data flow of the system terminates within some unit in the system. At the end of the section we will describe how to specify an open loop or a closed loop system.

There are two kinds of statements: send statement and call statement. A send statement consists of a left part followed by a colon : and a right part. The left part must be a system variable which has been defined in the second part of the system The right part can be either a system variable or an input variable of a unit which has been defined in the first part of the system. When we want to use a variable in a unit, we need first give the name of the unit, then the name of the variable. For example, F control_force means that the variable control_force is a variable of the unit F. The colon : is considered to be an operator setting up a communication channel. For example,

```
x : op x
```

This statement says that the value of the system variable x provides the value for the input variable x of the unit op. Please note that a variable is an input variable or an output variable depending on its position in the statement. If a variable appears in the left part of ":" , it is an output variable. If a variable appears in the right part of ":", it is an input variable. So one may not put an input (output) variable of a unit in the right (left) part. But a system variable can be used both as an input variable and an output variable.

Now let us look at the call statement. A call statement starts with a unit name followed by a left brace {, a list of statements separated by commas, and a right brace. The list of statements in a call statement can contain only send statements. Each statement in the list has the following format:

```
name1 : name2
```

where name1 is an output variable name of the unit starting the call statement. Name2 is an input variable of a unit or a system variable. When name2 is a variable of a unit, it is given by the unit name and the variable name.

Example:

```
F {     control_force : op f,
        t             : t,
        x             : x
}
```

The above call statement says that the output variable control_force of the unit F provides input values for the input variable f of the unit op, the output variable t of the unit F provides input value for the system variable t, and the output variable x of the unit F provides input values for the system variable x. In this compound statement the system variables t and x are used as input variables.

Before ending this section let us talk about how to tell a system is an open loop or a closed loop. In a fuzzy inference system, if at least one system variable is used as both output variable (i.e. appear on the left part of a statement in the specifications) and input variable (i.e. appear on the right part of a statement of the specifications), then the system is a closed loop and the reserved word LOOP must be used. A fuzzy inference system is an open loop if it is not a closed loop, i.e. any system variable is used only once either as an input variable or an output variable. In an open loop the reserved word DO must be used. It is the programmer's responsibility to make sure that the used reserved word is consistent with the data flow specifications.

Note: In the current implementation of FCL, the data flow in a fuzzy inference system contains only one main loop. Nested loops are prohibited.

Chapter 3 Graphic Representation

Instead of describing a fuzzy inference system by using FCL, one can describe a fuzzy inference system by a graphic. A graphic describing a fuzzy inference system consists of rectangles, wedges, and wires, which correspond to the three parts in a FCL description of the system: units, variables and data flow specifications.

Each rectangle in a graphic represents a Fide unit in the system. Input variables of a unit are represented by small horizon line segments on the left side of the rectangle. The line segments on the left of a rectangle are called inpins of the rectangle. Output variables of a unit are represented by line segments on the right side of the rectangle. The line segments on the right of a rectangle are called outpins. The orders of the pins (inpin or outpin) are the same as the orders of the variables (input variable or output variable) in the unit. A rectangle together with its pins is called a unit node and the inpins and outpins are referred as inpins and outpins of the node. On the left upper corner of a unit node is a small box which specifies the type of unit represented by the node. The possible unit types are FIU, FOU and FEU.

Each wedge represents an occurrence of a system variable. If a wedge points to the right, the wedge represents a system variable used as an input variable. If a wedge points to the left, the wedge represents a system variable used as an output variable. A wedge pointing to the right is called an invar node and the sharp end of the wedge is called the outpin of the invar node. A wedge pointing to the left is called an outvar node and the sharp end of the wedge is called the inpin of the outvar node.

Each wire connecting the pin of a var node and a pin (inpin or outpin) of a unit node or connecting a pin of a unit node and a pin of an another unit node represents a communication channel (data flow connection), which corresponds to a data flow specification in the FCL representation of the system. Such a wire is called a connection (between nodes). The variable name(s) of a unit(s) which participates the data flow connection is not explicitly given in the graphics. But the order of the pins in a unit node implicitly specifies the variable in the unit since the pins on a unit node have the same order as the variables in the unit represented by the node.

A graphic representation can be created by using the graphic editor of Fide or generated from a FCL representation. On the other side, a graphic representation of a system can also be converted to an FCL form of the system.

Appendix Input File Format

Overview

In Composer, before executing a fuzzy inference system, if the user specifies an input file "XXXX.SIN", then the values for input variables will come from the file "XXXX.SIN". Suppose we have a fuzzy inference system with three (system level) input variables: `x1`, `x2`, and `x3`.

If a fuzzy inference system is a closed loop, that is, at least one output variable is used as input, then the input file is read only once (read as initial values). If the system is an open loop, then the input file is read for every step of execution.

The following is the format of the input file.

Input File Format

Filename extension: `SIN`

File Format:

The format is simple. The values for the variables are listed in the same order as the variables' order in the composer ".lnk" file. The separator between the values may be either a space or a new line.

As an example, suppose the lnk file has three input variables: `x1`, `x2`, `x3`, and we want the values for these variables to be 0, 1, 2 at the first step, and 5, 6, 7 at the second step. We can write this input file as follows:

```
0  1  2
5  6  7
```

Since a space can be used to distinguish between values for steps, we can also write this input file as :

```
0  1  2  5  6  7
```

Introduction

This manual contains definitions of all Fide library routines and example program code which illustrates how to use these routines. To understand the examples used in this manual, basic C programming knowledge is required. Here is a summary of the chapters in this manual:

Fuzzy Inference Unit introduces the concept of the fuzzy inference unit and gives the data structures used by the Fide library.

How to Use the Fide Library discusses how to embed a fuzzy inference unit into an application program.

Functions is the chapter to reference for all Fide library functions. This chapter begins with a short table summarizing all function names and their purpose.

Examples gives various examples of C code programs to show how to use the Fide library.

The Fide library consists of a header file, "fiu.h", and the following library files for different environments:

File	Environment	Model
fiulibts.lib	Turbo C or Borland C	Small
fiulibtm.lib	Turbo C or Borland C	Medium
fiulibtc.lib	Turbo C or Borland C	Compact
fiulibtl.lib	Turbo C or Borland C	Large
fiulibth.lib	Turbo C or Borland C	Huge
fiulibms.lib	Microsoft C	Small
fiulibmm.lib	Microsoft C	Medium
fiulibmc.lib	Microsoft C	Compact
fiulibml.lib	Microsoft C	Large
fiulibmh.lib	Microsoft C	Huge

Chapter 1 Fuzzy Inference Unit

A fuzzy inference unit can be considered a box with several input and output variables and an internal mechanism. When a fuzzy inference unit is executed, its internal mechanism generates output values corresponding to the input values. Details of the internal mechanism are discussed in the **FIL Reference Manual**.

The value of an input variable belongs to a set called the range interval of the variable, specified by its minimum and maximum value. The range interval is further divided into subintervals of the same length, called the step length of the variable. The minimum, maximum and step length of a variable are given in the fuzzy inference unit program. (cf. **Fide Inference Language**). The value of the input variable is replaced by the ordinal number (counting from left to right and starting from 0) of the subinterval, which the input value belongs to. The ordinal number is called the normalized value of the input value. To avoid confusion, the input value itself will be called the **real value**. The correspondence between an input value and its normalized value can be written as:

normalized value = (real value - minimum)/step length

Note:

Either real or normalized values can be input into a fuzzy inference unit, depending upon the user's application.

For a C program to utilize a fuzzy inference unit, the FIU is specified by its name but without an extension. One or more fuzzy inference units can be used by a C program. When an FIU is compiled, three object files are generated by the Fide compiler. The object files have the same name as the fuzzy inference unit, but each has a unique extension. If *XXXXXX* is the name of a fuzzy inference unit, then the three object files are:

XXXXXX.MAP the map file
XXXXXX.COD the code file
XXXXXX.MBF the membership function file

The map file contains general information about the fuzzy inference unit. The code file contains inference codes for the fuzzy inference unit which drives the inference mechanism. The membership function file contains membership function data. This data is used by the fuzzy inference mechanism.

Data Structure for Fuzzy Inference Unit

A fuzzy inference unit should be initialized before it can be utilized. The initializing data that specifies a fuzzy inference unit are shown in Table 1. The data are stored separately in the data structure.

Table 1.

Data Structure	Content
FIU	General information including pointers to the related data structures
FIU_CODE	Inference codes
FIU_MBF	Membership functions
FIU_SGT	Singletons
FIU_IO	I/O buffers

Up to four fuzzy inference units are simultaneously accessible though a global array of four FIU structures. However, the data structure FIU_CODE, FIU_MBF and FIU_IO should be allocated by the C application (see sample1.c).

Note:

(1) The name of the data structures in the above table are defined as global names and should not be redefined anywhere else in the C application.

(2) If more than four fuzzy inference units are needed, the C application should allocate FIU structures, and call the *fiupcy* function (see sample4.c).

Chapter 2 How To Use The Fide Library

An application programmed in the C language can utilize fuzzy inference units compiled with the Fide Compiler. This is made possible by definitions and implementations which the Fide library provides. Before utilizing a fuzzy inference unit in an application program, the name of the fuzzy inference unit should be associated with an FIU data structure. This is done by calling the library function *fiuopen*, which makes the specified fuzzy inference unit accessible. The *fiuopen* function returns a fuzzy inference unit handle to the caller. The *fiuopen* function also returns the size of memory required for executing the specified fuzzy inference unit.

The application program, having the required memory allocated, can initialize an executor for the specified fuzzy inference unit by calling the library function *fiuinit*.

An initialized fuzzy inference unit can be executed repeatedly until it is closed by calling *fiuclose*. In each execution, the input values (of the input variables) are assigned to their respective cells in the I/O buffer. To enable this assignment, the Fide library provides the functions *fiuinvar*, *fiuoutvar*, *fiuinval* and *fiuoutval* that return pointers to the cells of specified input and output variables.

When input values are placed in the I/O buffer, the user's program can execute the fuzzy inference unit by calling the *fiuexec* or *fiupexec* function. These functions returns a succeed or fail message to the caller, and if it is successful, the output values generated by the execution are placed in the I/O buffer.

Listed in table 2 are the typical steps for embedding a fuzzy inference unit in an application program (see sample1.c):

Table 2.

STEP	Description
1	Use *fiuopen* function to open an existing fuzzy inference unit, and get the fuzzy inference unit handle, buffer size of code, membership functions and I/O
2	Allocate memory for storing code, membership functions and I/O data of fuzzy inference unit
3	Use *fiuinit* function to pass the pointers (to the code, membership functions and I/O memory) to an opened fuzzy inference unit
4	Use *fiuinvar* or *fiuinval* to get the pointers to the input variable
5	Use *fiuoutvar* or *fiuoutval* to get the pointers to the output variables
6	Set input values of the fuzzy inference unit
7	Use *fiuexec* or *fiurec* to execute the fuzzy inference unit
8	Get inference result
9	Repeat step 6 if necessary
10	Use *fiuclose* function to release a fuzzy inference unit

The Fide library provides the *fiucpy* function for two primary reasons: 1.) If more than four fuzzy inference units are used, or 2.) each fuzzy inference unit is to be handled directly. The *fiucpy* function can copy an initialized fuzzy inference unit to a user's pre-allocated fuzzy inference unit buffer. A copied fuzzy inference unit is then executed by calling the function *fiupexec* or *fiuprec*.

Listed in table 3 are the typical steps for using a pre-allocated fuzzy inference unit.

Table 3.

STEP	Description
1	Allocate memory for the pre-allocated fuzzy inference unit.
2	Follow step 1 to step 5 in **Table 2** to initialize a fuzzy inference unit.
3	Use the *fiucpy* function to copy the initialized fuzzy inference unit to the pre-allocated fuzzy inference unit.
4	Now the initialized fuzzy inference unit can be released safely, by calling the *fiuclose* function, if necessary.
5	Set input values for the pre-allocated fuzzy inference unit.
6	Use *fiupexec* or *fiuprec* to execute the fuzzy inference unit.
7	Get the inference result from pre-allocated fuzzy inference unit.
8	Repeat step 5 if necessary
9	Release all memory resources for pre-allocated fuzzy inference units.

Chapter 3 Functions

A list of the functions is given in table4 and the functions are classified by task in Table 5.

Table 4.

Routine	Description
fiuclose	Disable a fuzzy inference unit.
fiucpy	Copy a fuzzy inference unit to a pre-allocated fuzzy inference unit.
fiuexec	Execute a fuzzy inference unit.
fiuinit	Initialize a fuzzy inference unit.
fiuinval	Get a pointer to the real input variable in the I/O buffer.
fiuinvar	Get a pointer to the normalized input variable in the I/O buffer.
fiuopen	Open a fuzzy inference unit.
fiuoutval	Get a pointer to the real output variable in the I/O buffer.
fiuoutvar	Get a pointer to the normalized output variable in the I/O buffer.
fiupexec	Execute a pre-allocated fuzzy inference unit.
fiuprec	Execute a pre-allocated fuzzy inference unit and record the result.
fiurec	Execute a fuzzy inference unit and record the result.

Table 5.

Task	Routine
Initialize a fuzzy inference unit.	*fiuinit*
Open a fuzzy inference unit	*fiuopen*
Get a pointer to the normalized I/O variable	*fiuinvar, fiuoutvar*
Get a pointer to the real I/O variable	*fiuinval, fiuinval*
Copy a fuzzy inference unit	*fiucpy*
Execute a fuzzy inference unit.	*fiuexec, fiupexec*
Execute a fuzzy inference unit and record the result.	*fiurec, fiuprec*
Disable a fuzzy inference unit.	*fiuclose*

fiuclose

PURPOSE

Use the *fiuclose* function to disable a fuzzy inference unit

SYNTAX

```
int     fiuclose(fiu)

int     fiu     fuzzy inference unit handle
```

EXAMPLE CALL

```
ret fiuclose(p_fiu);
```

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuclose* function disables the fuzzy inference unit specified by the argument *fiu*. Any memory allocated for the specific fuzzy inference unit is no longer in use.

RETURNS

If *fiu* does not specify a fuzzy inference unit, the *fiuclose* function returns -1 as an error code, otherwise, the returned value is zero.

fiucpy

PURPOSE

Use *fiucpy* to copy an initialized fuzzy inference unit to a pre-allocated fuzzy inference unit.

SYNTAX

```
int   fiucpy(fiud, fius)

FIU  *fiud      the pointer to the pre-allocated fuzzy inference unit
int   fius      the handle of the fuzzy inference unit to be copied
```

SAMPLE CALL

```
ret = fiucpy(&fiu5, fius);
```

INCLUDE

"fiu.h"

DESCRIPTION

The *fiucpy* function copies the initialized fuzzy inference unit specified by the argument *fius* to the pre-allocated fuzzy inference unit specified by the argument *fiud* .

RETURNS

The *fiucpy* function returns zero to the caller upon successful completion, otherwise, it returns a non-zero integer as an error code in the following cases: (1) If the argument *fius* does not specify an initialized fuzzy inference unit, the returned error code is -1. (2) If the argument *fiud* is a NULL pointer, the returned error code is -2.

fiuexec

PURPOSE

Use *fiuexec* to execute the fuzzy inference engine in a fuzzy inference unit.

SYNTAX

int fiuexec(fiu)

int fiu fuzzy inference unit handle

EXAMPLE CALL

ret = fiuexec(p_fiu);

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuexec* function executes the fuzzy inference unit specified by the argument *fiu* .

RETURNS

The *fiuexec* function returns zero to the caller upon successful completion, otherwise, it returns a non-zero integer as an error code in the following cases:

If the argument *fiu* does not specify an initialized fuzzy inference unit, the returned error code is -1.

If the execution is stopped by a zero divisor, the returned error code is a positive integer, and the number is the ordinal number of the inference code that caused the error.

fiuinit

PURPOSE

Use the *fiuinit* to initialize an executor of an opened fuzzy inference unit.

SYNTAX

int fiuinit(fiu, code, mbf, iobuf)

int fiu	fuzzy inference unit handle
unsigned char *code	pointer to the codes
unsigned char *mbf	pointer to the membership functions
unsigned char *iobuf	pointer to the I/O buffer

EXAMPLE CALL

ret = fiuinit(p_fiu, code_buff, mbf_buff, io_buff);

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuinit* function initializes an executor for the fuzzy inference unit specified by the argument *fiu* by reading the object files. The pointer arguments *code*, *mbf* and *iobuf* point to the memory for the inference codes, the membership functions and the I/O buffer of the fuzzy inference units, respectively.

RETURNS

The *fiuinit* function returns 0 to the caller if no error has occurred.

The *fiuopen* function returns a non-zero integer as an error code in response to the following:

If *fiu* does not specify an opened fuzzy inference unit, the returned error code is
-1.

If some of the arguments *code*, *iobuf* and *mbf* are NULL pointers, the returned error code is the sum of the following values:

08	argument *code* is a NULL pointer
16	argument *mbf* is a NULL pointer
32	argument *iobuf* is a NULL pointer

If one or more object files can not be opened, the error code is the sum of the following values:

01	code file can not be opened
02	membership function file can not be opened
04	map file can not be opened

fiuinval

PURPOSE

Use the *fiuinval* to get the I/O buffer address pointer to a given real input variable.

SYNTAX

```
float  *fiuinval(fiu,vname)

int    fiu     fuzzy inference unit handle
char   *vname  name of an input variable
```

EXAMPLE CALL

```
volt = fiuinval(m_fiu,"voltage");
```

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuinval* function returns a pointer to the cell of an input variable of the fuzzy inference unit specified by the fuzzy inference unit handle argument *fiu*. The name of the input variable is specified by the string argument *vname*. The cell is the real value field in the I/O buffer data structure of the fuzzy inference unit.

RETURNS

If *fiu* does not specify an opened fuzzy inference unit, or the specified input variable name is not found in the map file of that unit, the returned pointer is NULL.

fiuinvar

PURPOSE

Use the *fiuinvar* to get the I/O buffer address pointer to a given normalized input variable.

SYNTAX

```
unsigned char  *fiuinvar(fiu,vname)

int    fiu     fuzzy inference unit handle
char   *vname  name of an input variable
```

EXAMPLE CALL

```
speed = fiuinvar(p_fiu, "p_velocity");
```

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuinvar* function returns a pointer to the cell of an input variable of the fuzzy inference unit specified by the fuzzy inference unit handle argument *fiu*. The name of the input variable is specified by the string argument *vname*. The cell is the normalized value field in the I/O buffer data structure of the fuzzy inference unit.

RETURNS

If *fiu* does not specify an opened fuzzy inference unit, or the specified input variable name is not found in the map file of that unit, the returned pointer is NULL.

fiuopen

PURPOSE

Use *fiuopen* to open an existing fuzzy inference unit.

SYNTAX

```
int    fiuopen(fiup, name, codesize, iobuflen, mbfsize, mood)
```

int	*fiup	pointer to the handle of the fuzzy inference unit
char	*name	the name of the fuzzy inference unit
long	*codesize	returns the memory size of the inference codes
long	*mbfsize	returns the memory size of the membership functions
long	*iobuflen	returns the memory size of the I/O buffer
int	mood	(reserved)

EXAMPLE CALL

```
ret = fiuopen(&p_fiu, "pend", &rule_size, &io_size, &mbf_size, 0);
```

INCLUDE

"fiu.h"

DESCRIPTION

Specifies the fuzzy inference unit and stores necessary information. To do this, it reads the object files of the fuzzy inference unit, calculates the memory size for the inference codes, the membership functions and the I/O buffer of the fuzzy inference unit It then stores the calculated sizes in the cells pointed to by the pointer arguments *codesize*, *mbfsize* and *iobuflen*, respectively.

RETURNS

If the operation is successful, *fiuopen* stores the fuzzy inference unit handle in the cell pointed to by the pointer argument *fiu* and returns 0 to the caller. In the following cases, the *fiuopen* function returns a non-zero integer as an error code:

If the argument *fiup* is a NULL pointer, the returned error code is -1.

If there are too many (more than four) fuzzy inference units being opened, the returned error code is -2.

If one or more object files can not be opened, the returned error code is the sum of the following values:

01	code file can not be opened
02	membership function file can not be opened
04	map file can not be opened

Note: The argument *mood* is reserved. The only acceptable value is 0. If the value of *mood* is non-zero, an error code of -3 is returned.

fiuoutval

PURPOSE

Use the *fiuoutval* to get the I/O buffer address pointer to a given real output variable.

SYNTAX

```
float   *fiuoutval(fiu,vname)

int     fiu     fuzzy inference unit handle
char    *vname  name of an output variable
```

EXAMPLE CALL

```
current = fiuoutval(m_fiu,"current1");
```

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuoutval* function returns a pointer to the cell of an output variable. The cell belongs to the fuzzy inference unit specified by the fuzzy inference unit handle argument *fiu*. The name of the output variable is specified by the string argument *vname*. The cell is the real value field in the I/O buffer data structure of the fuzzy inference unit.

RETURNS

If *fiu* does not specify an opened fuzzy inference unit, or the specified output variable name is not found in the map file of that unit, the returned pointer is NULL.

fiuoutvar

PURPOSE

Use the *fiuoutvar* to get the I/O buffer address pointer to a given normalized output variable.

SYNTAX

```
unsigned char * fiuoutvar(fiu, vname)

int     fiu     fuzzy inference unit handle
char    *vname  name of an output variable
```

EXAMPLE CALL

```
acce = fiuoutvar(p_fiu, "accelerate");
```

INCLUDE

"fiu.h"

DESCRIPTION

The *fiuoutvar* function returns a pointer to the cell of an output variable. The cell belongs to the fuzzy inference unit specified by the fuzzy inference unit handle argument *fiu*. The name of output variable is specified by the string argument *vname*. The cell is the normalized value field in the I/O buffer data structure of the fuzzy inference unit.

RETURNS

If *fiu* does not specify an opened fuzzy inference unit, or the specified output variable name is not found in the map file of that unit, the returned pointer is NULL.

fiupexec

PURPOSE

Use *fiupexec* to execute a fuzzy inference unit which is pre-allocated .

SYNTAX

int fiupexec(pfiu)

FIU *pfiu pointer to a pre-allocated fuzzy inference unit

EXAMPLE CALL

ret = fiuexec(&fiu5);

INCLUDE

"fiu.h"

DESCRIPTION

The *fiupexec* function executes the pre-allocated fuzzy inference unit specified by the argument *pfiu* .

RETURNS

The *fiuexec* function returns zero to the caller upon successful completion. Otherwise, it returns a non-zero integer as an error code in the following cases:

If the argument *pfiu* does not specify an initialized fuzzy inference unit, the returned error code is -1.

If the argument *pfiu* is a NULL pointer, the returned error code is -1.

If the execution is stopped by zero division, the returned error code is a positive integer and the number is the ordinal number of the inference code that caused the error.

fiuprec

PURPOSE

Use *fiuprec* to execute a pre-allocated fuzzy inference unit and record the fuzzy inference result in a text file.

SYNTAX

```
int     fiuprec(fiup,fp)

FIU     *fiup    pointer to the pre-allocated fuzzy inference unit
FILE    *fp      pointer to the record file
```

EXAMPLE CALL

```
ret = fiuprec(&fiu5, rec_fp);
```

INCLUDES

```
<stdio.h>
"fiu.h"
```

DESCRIPTION

The *fiuprec* function executes the pre-allocated fuzzy inference unit which is specified by the argument *fiup*, and records the inference result in the text file specified by the argument *fp*.

RETURNS

The *fiuprec* function returns zero to the caller upon successful completion. Otherwise, it returns a non-zero integer as an error code in the following cases:

If the argument *fiup* does not point to an initialized fuzzy inference unit, the returned error code is -1.

If the argument *fp* is a NULL pointer, the returned error code is -2.

If the *fiuprec* function is stopped by zero division, the returned error code is a positive integer and the number is the ordinal number of the inference code that caused the error.

fiurec

PURPOSE

Use *fiurec* to execute a fuzzy inference unit and record the fuzzy inference result in a text file.

SYNTAX

```
int     fiurec(fiu,fp)

int     fiu     fuzzy inference unit handle
FILE  * fp      pointer to the record file
```

EXAMPLE CALL

```
ret = fiurec(p_fiu, rec_fp);
```

INCLUDES

```
<stdio.h>
"fiu.h"
```

DESCRIPTION

The *fiurec* function executes the fuzzy inference unit which is specified by the argument *fiu*. The *fiuprec* function also records the inference result in the text file specified by the argument *fp*.

RETURNS

The *fiurec* function returns zero to the caller upon successful completion. Otherwise, it returns a non-zero integer as an error code in the following cases:

If the argument *fiu* does not specify an initialized fuzzy inference unit, the returned error code is -1.

If the argument *fp* is a NULL pointer, the returned error code is -2.

If the *fiurec* function is stopped by zero divison, the returned error code is a positive integer and the number is the ordinal number of the inference code that caused the error.

Chapter 4 Examples

sample1.c

```
/*************************************************************************/
/* Name    : sample1.c                                                   */
/* purpose: Example for using the basic Fide library functions           */
/* Aptronix, Inc. 4-28-1992                                              */
/*************************************************************************/
#include <stdio.h>
#include <stdlih.h>
#include <mem.h>
#include "fiu.h"                        /* include the Fide header file     */

unsigned char *volt;                    /* pointer to output variable       */
unsigned char *theta, *omega;           /* pointer to input variable        */
long codesize,mbfsize,iobuflen;         /* size of buffers                  */
unsigned char * code_buff;              /* pointer to the code buffer       */
unsigned char * mbf_buff;               /* pointer to the MBFs buffer       */
unsigned char * io_buff;                /* pointer to the I/O buffer        */
int   fh;                               /* FIU handle                       */

void main(){
  /* open a fuzzy inference unit, "example1" */
  if(fiuopen(&fh,"example1",&codesize,&mbfsize,&iobuflen,0)!=0) exit(0);

  /* allocate memory for code, membership functions and I/O buffer */
  if(code_buff=(unsigned char *)malloc(codesize), code_buff==NULL) exit(0);
  if(mbf_buff=(unsigned char *)malloc(mbfsize),   mbf_buff==NULL) exit(0);
  if(io_buff=(unsigned char *)malloc(iobuflen), io_buff==NULL) exit(0);

  /* initialize an opened fuzzy inference unit */
  if(fiuinit(fh,code_buff,mbf_buff,io_buff,0)!=0) exit(0);

  /* get input and output variable pointers */
  volt = fiuoutvar(fh,"volt");
  theta = fiuinvar(fh,"theta");
  omega = fiuinvar(fh,"omega");

  /* set input values */
  *omega = 96;
  *theta = 185;
  printf("\ntheta=%03d,omega=%03d: ",*theta,*omega);

  /* execute the fuzzy inference unit */
  fiuexec(fh);

  /* get inference result */
  printf("volt=%3d \n",*volt);

  /* close fuzzy inference unit */
  fiuclose(fh);

  /* release memory */
  free(code_buff);
  free(mbf_buff);
  free(io_buff); }
```

sample2.c

```
/*************************************************************************/
/* Name    : sample2.c                                                   */
/* purpose: Example for using fiurac function                            */
/* Aptronix, Inc. 1-28-1992                                              */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
#include "fiu.h"                    /* include the Fide header file     */

unsigned char *volt;                /* pointer to output variable       */
unsigned char *theta, *omega;       /* pointer to input variable        */
long codesize,mbfsize,iobuflen;     /* size of buffers                  */
unsigned char * code_buff;          /* pointer to the code buffer       */
unsigned char * mbf_buff;           /* pointer to the MBFs buffer       */
unsigned char * io_buff;            /* pointer to tha I/O buffer        */
int   fh;                           /* FIU handle                       */
FILE  *fp;                          /* record file pointer              */

void main(){
  int i;
  /* open a fuzzy inference unit, "example1" */
  if(fiuopen(&fh,"example1",&codesize,&mbfsize,&iobuflen,0)!=0) exit(0);

  /* allocate memory for code, membership functions and I/O buffer */
  if(code_buff=(unsigned char *)malloc(codesize), code_buff==NULL) exit(0);
  if(mbf_buff=(unsigned char *)malloc(mbfsize),  mbf_buff==NULL) exit(0);
  if(io_buff=(unsigned char *)malloc(iobuflen), io_buff==NULL) exit(0);

  /* initialize an opened fuzzy inference unit */
  if(fiuinit(fh,code_buff,mbf_buff,io_buff,0)!=0) exit(0);

  /* get input and output variable pointers */
  volt = fiuoutvar(fh,"volt");
  theta = fiuinvar(fh,"theta");
  omega = fiuinvar(fh,"omega");

  /* open record Fide */
  fp = fopen("example1.rec","w");

  /* set input values */
  *omega = 96;                              /* set input value omega  */
  for ( i = 30; i<240 ; i+=30){
        *theta = (unsigned char) i;         /* set output value theta */

       /* execute the fuzzy inference unit and record inference result */
        fiurec(fh,fp);
  }

  /* close the fuzzy inference unit */
  fiuclose(fh);

  /* release memory */
  free(code_buff);
```



```
  free(mbf_buff);
  free(io_buff);

  /* close record file */
  fclose(fp);
}
```

sample3.c

```
/*************************************************************************/
/* Name   : sample3.c                                                    */
/* purpose: Example for using real data in an application                */
/* Aptronix, Inc. 4-28-1992                                              */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
#include "fiu.h"                        /* include the Fide header file    */

float  *I;                              /* pointer to output variable      */
float  *V1,*V2;                         /* pointer to input variable       */
long codesize,mbfsize,iobuflen;         /* size of the buffers             */
unsigned char * code_buff;              /* pointer to the code buffer      */
unsigned char * mbf_buff;               /* pointer to the MBFs buffer      */
unsigned char * io_buff;                /* pointer to the I/O buffer       */
int   fh;                               /* FIU handle                      */

void main(){
  /* open a fuzzy inference unit, "example2" */
  if(fiuopen(&fh,"example2",&codesize,&mbfsize,&iobuflen,0)!=0) exit(0);

  /* allocate memory for code, membership functions and I/O buffer */
  if(code_buff=(unsigned char *)malloc(codesize), code_buff==NULL) exit(0);
  if(mbf_buff=(unsigned char *)malloc(mbfsize),  mbf_buff==NULL) exit(0);
  if(io_buff=(unsigned char *)malloc(iobuflen), io_buff==NULL) exit(0);

  /* initialize an opened fuzzy inference unit */
  if(fiuinit(fh,code_buff,mbf_buff,io_buff,0)!=0) exit(0);

  /* get real input and output variable pointers */
  I  = fiuoutval(fh,"current");
  V1 = fiuinval(fh,"voltage1");
  V2 = fiuinval(fh,"voltage2");

  /* set input values */
  *V1 = -3.2;
  *V2 = 4.8;
  printf("\nV1=%f,V2=%f: ",*V1,*V2);

  /* execute the fuzzy inference unit */
  fiuexec(fh);

  /* get the inference result */
  printf("I=%f \n",*I);

  /* close the fuzzy inference unit */
  fiuclose(fh);

  /* release memory */
  free(code_buff);
  free(mbf_buff);
  free(io_buff);
}
```


sample4.c

```
/*************************************************************************/
/* Name   : sample4.c                                                    */
/* purpose: Example for using pre-allocated fuzzy inference unit         */
/* Aptronix, Inc. 4-28-1992                                              */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
#include "fiu.h"                        /* include the Fide header file    */

unsigned char *volt;                    /* pointer to output veriable      */
unsigned char *thete, *omege;           /* pointer to input variable       */
long codesize,mbfsize,iobuflen;         /* size of buffers                 */
unsigned char *code_buff;               /* pointer to the code buffers     */
unsigned char *mbf_buff;                /* pointer to the MBFs buffers     */
unsigned char *io_buff;                 /* pointer to the I/O buffers      */
int   fh;                               /* FIU handle                      */
FIU   pfiu;                             /* pre-allocated fiu               */

void main(){
  /* open a fuzzy inference unit, "example" */
  if(fiuopen(&fh,"example",&codesize,&mbfsize,&iobuflen,0)!=0) exit(0);

  /* allocate memory for code, membership functions and I/O buffer */
  if(code_buff=(unsigned char *)malloc(codesize), code_buff==NULL) exit(0);
  if(mbf_buff=(unsigned char *)malloc(mbfsize),  mbf_buff==NULL) exit(0);
  if(io_buff=(unsigned char *)malloc(iobuflen), io_buff==NULL) exit(0);

  /* initialize an opened fuzzy inference unit */
  if(fiuinit(fh,code_buff,mbf_buff,io_buff,0)!=0) exit(0);

  /* get input and output variable pointers */
  volt = fiuoutvar(fh,"volt");
  thete = fiuinvar(fh,"theta");
  omega = fiuinvar(fh,"omega");

  /* copy initialized "example" fiu to e pre-allocated fiu */
  fiucpy(&pfiu, fh);

  /* release fiu */
  fiuclose(fh);

  /* set input values */
  *omega = 96;
  *theta = 185;

  /* execute pre-allocated fuzzy inference unit */
  fiupexec(&pfiu);
  /* get inference result */
  printf("\ntheta=%03d,omega=%03d: ",*thete,*omege);
  printf("volt=%3d \n",*volt);

  /* release memory */
  free(code_buff);
```

```
    free(mbf_buff);
    free(io_buff);
}
```

Index